Tierless Web Programming in the Large

Gabriel Radanne
University of Freiburg
Germany
radanne@informatik.uni-freiburg.de

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

ABSTRACT

Tierless Web programming languages allow combining client-side and server-side programming in a single program. This allows defining 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 modern ML languages, the module system is a layer separate from the expression language.

Eliom is an extension of OCaml for tierless Web programming which provides type-safe communication and an efficient execution model. In this article, we present how the Eliom module system combines the flexibility of tierless Web programming languages with a powerful module system, thus providing good support for abstraction, modularity and separate compilation. We also show that we can provide all these advantages while providing seamless integration with OCaml and its ecosystem.

CCS CONCEPTS

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

KEYWORDS

Web; client/server; OCaml; ML; Eliom; functional; module

ACM Reference Format:
Gabriel Radanne and Jérôme Vouillon. 2018. Tierless Web Programming in the Large. In *The 2018 Web Conference Companion, April 23–27, 2018, Lyon, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3184558.3185953

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 just 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 cannot 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 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 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. Even worse, almost no tierless programming languages support separate compilation. Separate compilation, or its weaker form incremental compilation, is an essential-productivity boost for programmers working on medium to large scale software.

To solve these problems, we propose to leverage a well-known tool: ML-style modules. 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 expressions 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 [19] uses this notion of collection of declarations

WWW'18 Companion, April 23–27, 2018, Lyon, France

ACM ISBN 978-1-4503-5640-4/18/04.
https://doi.org/10.1145/3184558.3185953

(called *structures*) 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 [16] and provides support for datatype abstraction [8, 17], 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 architect 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 [32] (also known as first-class modules). One distinctive feature is that modules in OCaml are runtime entities. In contrast to systems such as MLton [23], they are not eliminated at compile time.

1.2 Eliom

Eliom [29, 30] 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 [2, 11] project, which also includes the compiler js_of_ocaml [38], 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 [25]; a powerful session mechanism and an advanced *service identification mechanism* [1]. The Ocsigen project started in 2004, as a research project, with the goal of building a complete industrial-strength framework.

1.3 Modules for tierless web programming

All of the modules and libraries in Ocsigen, and in particular in the Eliom framework, are implemented on top of a core language described by Radanne et al. [30]. 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 an efficient execution model. We introduce additional properties that 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 concerns 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.

Modularity and separate compilation. Far from simple websites, modern Web applications are complex programs that rival regular desktop programs in size. Modularity and separate compilation are essential tools to make programmers productive for large applications. Eliom is the only tierless programming language that provides static slicing, efficient execution and separate compilation.

In the rest of this article, we present how Eliom module system provides abstraction and modularity for tierless applications. As a guiding example for our exploration of the Eliom language, we consider the case of a commenting system, as can be found on numerous websites. A comment is a piece of HTML written by a user and identified by a unique identifier. Such a comment library features both server aspects (storing and rendering the comments) and client interactions (browsing and searching comments).

We first give a reminder of the OCaml module system (Section 2) and small-scale tierless programming (Section 3). We then present the Eliom module language (Section 4) and the challenges regarding its implementation in (Section 5). Finally, we give a quick comparison with existing work in Section 6.

2 OF COMMENTS AND CAMELS
A SHORT INTRODUCTION TO OCAML MODULES

The OCaml module system forms a second language separate from the expression language. While the language of expressions allows programming "in the small", the module language allows programming "in the large". The ML flavor of module systems, which OCaml is part of, significantly extend usual module languages by providing module types (called signatures) and functions from modules to modules (called functors). The module system is implicitly used for any kind of OCaml or Eliom programming: Each `.ml` and `.eliom` file form a structure containing the list of declarations included in the file. It is also possible to specify a signature for such module by adding a `.mli` or `eliomi` file.

We can do a lot more with OCaml modules. For example, let us say we are writing an `HTML` library. We want to gather the event related attributes in a single module. We can easily do so with the following construction.

```
module On = struct
  let click = ...
  let keypress = ...
end
```

These functions can then be used through qualified accesses:

```
open Html
let mywidget = div ~a:[On.click myclickhandler] [ ... ]
```

Some users of our HTML library may want to experiment with new, custom-made HTML elements. They can easily do so by extending the `Html` module:

```
module HtmlPlus = struct
  include Html
  let blink elems = ...
end
```

Here, we declare a new module, `HtmlPlus`, in which we *include* `Html` and define the new `blink` function. The include operation simply takes all the fields of a module and adds them to the enclosing module. This way, we obtain a new module `HtmlPlus` which can be used anywhere `Html` can, but also includes the new function.

2.1 Abstraction and encapsulation

We now want to build a simple library to handle internet comments. In our library, comments are pieces of HTML (constructed with the `Html` module) identified by a unique number. We are not sure yet if we should use simple sequential IDs, date-base IDs or something else like UUIDs and Hashids [12]. Fortunately, we do not have to make this decision immediately! All we need in order to write the rest of our engine is an interface for creating and using identifiers. We can declare such an interface in OCaml using a *signature*. Below, we declare the `ID` signature describing what a module implementing unique identifiers should look like.

```
module type ID = sig
  type t (* type of ids *)
  val compare : t -> t -> int (* Compare ids *)
  val create : unit -> t (* Create fresh ids *)
  val to_string : t -> string (* show ids *)
end
```

We can then create various modules implementing this specification. Here we declare the modules `SequentialID` and `DateID`. We can then switch one module for the other easily.

```
module SequentialID: ID = struct
  type t = int
  (* ... *)
end
```

```
module DateID: ID = struct
  type t = date
  (* ... *)
end
```

One important thing to note here is that, to the outer world, these two modules have exactly the same type and can not be distinguished. The type that implements the identifiers in the `ID` signature is abstract: its implementation is only visible inside the module and can not be used outside. It is also useful to note that such abstraction can be provided after the fact. Declaring a module and abstracting its interface are completely distinct operations.

Hiding the internal details of our `ID` modules is not only useful for modularity: it also allows to enforce abstraction boundaries. For example in the case of `SequentialID`, it is impossible to inadvertently use the ID as an integer, since the fact that it is an integer is not revealed! We can use this fact to enforce numerous complex properties, as we see in the next section.

2.2 Functors

To implement our comment system, we sometimes need to find comments by their ID. The idiomatic OCaml solution is to use maps, also called dictionaries. Such maps are implemented with Binary Search Trees which require a comparison function on the keys of the map. `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 dictionaries whose keys are of the type `t` in the provided module. In Fig. 2, we use this functor to create the `IDMap` module which defines dictionaries with IDs as keys. This is very easy, since the `ID` signature is already a super-set of the `COMPARABLE` signature. We then define `register`, a function which associates a fresh id to a comment `c`.

The `Map.Make` functor uses abstraction in two important ways. First, since the type of the map is abstract, it is impossible to modify it through means not provided by the module. In particular, this enforces that the binary tree is always balanced. Second, since the comparison function is provided in advance by the argument of the functor, it is impossible to mix different comparison functions

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

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

Figure 1: the `Map` module

```
module TheID = DateID (* The ID of our choice *)
module IDMap = Map.Make(TheID)
let register c map : Html.t IDMap.t =
  IDMap.add (TheID.create ()) c map
```

Figure 2: Dictionaries from IDs to comments

by mistake. Indeed, application of the functor to different modules would yield different types of maps.

2.3 Going further

This was just a taste of modules. For a longer (and better) introduction to modules, please consult the OCaml manual [18] or the Real World OCaml book [22].

3 TIERLESS WIDGETS

Until now, we presented how to write various elements of libraries useful for our comment system. For this purpose, we leveraged the power of the OCaml module system in various ways. We now want to write the widget that presents a comment. For this purpose, we need to define both client and server code, along with some client-server communication, which is precisely where tierless languages shine. Through this example, we provide a quick overview of the Eliom expression language, along with some basic associated concepts regarding tierless web programming.

3.1 Sections

Section annotations allow the programmer to specify where a declaration should be executed. The programmer can specify whether a declaration is to be performed on the server or on the client as follows:

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

In particular, sections allow to group related code in the same file, regardless of where it is executed. In the rest of this article, we use the following color convention: client is in **yellow**, server is in **blue** and mixed is in **green**. Colors are however not mandatory to understand the rest of this article.

3.2 Client fragments

While section annotations allow programmers to gather code across locations, they don't allow convenient communication. For this purpose, Eliom allows to include client-side expression inside a server section: an expression placed inside `[%client ... ]` will be computed on the client when the page is received; but the eventual client-side value of the expression can be passed around immediately as a black box on the server. These expressions are called client *fragments*.

In the example below, 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 variable `x` is only usable server-side, and has type `int fragment` which should be read "a fragment containing some integer". The value inside the client fragment cannot be accessed on the server.

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

3.3 Injections

Fragments allow programmers to manipulate client values on the server. We also need the opposite direction. 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. The value inside the client fragment is extracted by `~%x`, whose value is 4 here.

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

3.4 Comment widget

These three constructions are sufficient to create complex client-server interactions. Here, we use them to build a very simple widget to show one comment. Our widget, implemented by the function `make_comment` shown below, has the additional feature that it will hide the content of the comment when the user clicks on it. We also want the HTML to be generated server-side and sent to the client as a regular HTML page. This allows the comments to be accessible even when JavaScript cannot run. The implementation, the interface and the produced HTML fragment are shown in Fig. 3.

In order to implement our comment widget, we use an HTML DSL [37] that provides combinators such as `div` and `a_onclick` (which respectively create an HTML tag and an HTML attribute). The `~a` is the OCaml syntax for named arguments. Here, it is used for the list of HTML attributes. We first create a `p` element which contain the text of the comment and a unique id. The text is included in a `div` which represents the comment. We then use a handler listening to the `onclick` event: since clicks are performed client-side, this handler needs to be a client function inside a fragment. Inside the fragment, an injection is used to access the argument `id` which contains the identifier of the comment. We then use this identifier to fetch the correct element and toggle the "hidden" CSS property, which hides it.

As we can see, this type does not expose the internal details of the widget's behavior. In particular, the communication between server and client does not leak in the API: This provides proper encapsulation for client-server behaviors. Furthermore, this widget is easily composable: the embedded client state cannot affect nor be affected by any other widget and can be used to build larger widgets.

```
let%server make_comment commentid =
  let content =
    p ~a:[a_id commentid] [text (Comment.get commentid)]
  in
  let author = text ("Author: " ^ Comment.author commentid) in
  let handler = [%client fun _ ->
    let elem = get_element_by_id ~%commentid in
    Css.toggle_hidden elem]
  in
  div ~a:[On.click handler] [author; content]
```

```
val%server make_comment : id -> Html.element
```

```
<div>
  Author: Me
  <p id=1337>Eliom is great!</p>
</div>
```

Figure 3: The comment widget

3.5 Notes on semantics

In the examples above, we showed that we can interleave client and server expressions and communications in fairly arbitrary manners. This would be costly if the communication between client and server were done naively.

Instead, the server only sends data once when the Web page is sent. In particular, in the comment widget presented above, the id of the comment is not sent for each click. This is made possible by the fact that client fragments are not executed immediately when encountered inside server code. Intuitively, the semantics is the following. When the server code is executed, the encountered client code is not executed right away; instead it is just registered for later execution once the Web page has been sent to the client. Only then is the client code executed. We also guarantee that client code, be it either client sections or fragments, is executed in the order that it was encountered on the server. This presentation might makes it seem as if we dynamically create the client code during execution of the server code. This is not the case. Like OCaml, Eliom is statically compiled and separates client and server code at compile time. During compilation, we statically extract the code included inside fragments and compile it as part of the client code to JavaScript. This allows us to provide both an efficient execution scheme that minimizes communication and preserve side-effect orders while still presenting an easy-to-understand semantics. We also benefits from optimizations done by the js_of_ocaml compiler, thus producing efficient and compact JavaScript code.

3.6 Further reading

We only gave a brief overview of what can be done with the new language constructs introduced by Eliom. Radanne et al. [29] present numerous advanced examples which cover many Web programming idioms such as HTML, RPCs, broadcasts and other communication patterns. More formally, Radanne et al. [30] give a detailed account of the type system, the semantics and the compilation scheme for the Eliom expression language. For the rest of this article, we focus on large-scale tierless Web programming through modules.

4 TIERLESS MODULAR PROGRAMMING

We are now equipped with two tools. On one hand, we have a rich and expressive non-tierless module system, as presented in Section 2, which provides abstraction and modularity at the library

level. On the other hand, we have a powerful tierless programming language, as presented in Section 3, which allows us to describe sophisticated client-server behaviors. In this section, we present how we can bring those two tools together and reap the numerous benefits of the OCaml module system in a tierless setting.

4.1 Interaction with OCaml

Web programming is never only about the Web. Web programmers needs external libraries and a rich ecosystem that can not be provided by a fresh new language. Before writing complex tierless programs, let us see how Eliom can leverage the OCaml ecosystem almost transparently.

Integration with the OCaml language is provided through the use of a third location called **base**. Code located on base can be used both on the client and on the server.

```
let%base f x = "Hello "^x^"!"
let%client a = f "client"
let%server b = f "server"
```

Eliom-specific features such as fragments and injections are not allowed inside base code. In fact, base code corresponds exactly to OCaml code. This equivalence holds in theory but also in practice, meaning that any OCaml library compiled by the vanilla OCaml compiler can be directly reused by Eliom as being on the base location. This allows a very smooth integration with the OCaml ecosystem. Furthermore, a given OCaml library can be loaded either on base, on the client or on the server, depending on what the user wants. For example, an OCaml library manipulating file descriptors might be better kept only on the server in order to avoid misuse. The type-checker then raises an error if the library is mistakenly used on the client.

4.2 Modules and locations

As demonstrated in Section 4.1, OCaml modules, such as the `String` module taken from the standard library, are immediately available as Eliom modules located on base. We can also use such modules on the client or on the server.

```
module%base TextHtml = Html
module%server ServerHtml = TextHtml
let%client l = Html.p [Html.text "Hello client!"]
```

Locations are checked by the compiler. For example, using a server module on the client is forbidden.

```
let%client x = ServerHtml.text "hello client!" (* ✘ Error! *)
```

It is also possible to reuse OCaml module types freely. For example, we might want to define a client module `DomHtml` which shares the exact same API as the `Html` module, but is implemented using the Document Object Model that is available on the client. The type declaration for such a module would then be very simple, as shown below.

```
module%client DomHtml : Html.Signature = struct
  (* ... *)
end
```

We can easily declare a new structure completely on one location. The constraint is that all the fields on such modules, including submodules, should be on the same location. For example, a client structure can only contain fields that are declared on the client. The following piece of code declares a `JsMap` client module containing various fields and implementing a dictionary data-structure with JavaScript strings.

```
module%client JsMap : sig
  type key
  type 'a t

  val empty : 'a t
  val add : key -> 'a -> 'a t -> 'a t
end
```

We can also use functors in client and server code as we would in regular OCaml code. Consider the `JsMap` module above. The simplest way to obtain such a module would be to use the `Map.Make` functor presented in Section 2.2. We could for example write a `JsDate` module which uses JavaScript native support for dates. We can then obtain the `JsDateMap` module simply by applying `Map.Make` to the module `JsDate` defined in Fig. 4. As expected, the module we obtain is directly on the client. We can thus mix and match client and server modules using the tierless features and vanilla OCaml modules. This also works with all the other module features such as abstraction, high order functors and module inclusion. In all these cases, the Eliom typechecker ensures that modules always end up on the appropriate location.

```
module%client JsDate = struct
  type t = Js.date
  let compare x y = compare x##valueOf y##valueOf
    (** Compare by timestamp *)
end
module%client JsDateMap = Map.Make(JsDate)
```

Figure 4: Definition of `JsID` and `JsMap`

4.3 Mixed modules

Up until now, we only defined single-location modules, either base, client or server. It is natural to also want to write modules that contain base, client and server declarations. We call these modules "mixed".

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

Just like sections, mixed modules allow to group together declarations that are semantically related, regardless of client-server boundaries. However, combining type declarations with mixed modules and module signatures can provide even further benefits.

4.3.1 Encapsulation and Abstraction. A common idiom of web programming is to generate some HTML element on the server, add an `id` to it, and recover the element on the client through the `get_element_by_id` function. Indeed, this is exactly what we did in our comment widget in Section 3.4. This is so common, in fact, that it could be considered the "id design pattern". RPCs, channels and other communication APIs also follow the same mechanisms through the use of uniquely defined URLs. In all these cases, the means of identification for a given object is generally passed around explicitly, instead of being abstracted. Since client and server code are usually written separately, the programmer *must* expose the internal details to the outer world, including how to identify objects.

By combining tierless annotations and the abstraction capabilities provided by modules, we can recover that lost abstraction. Fig. 6 presents an API that encapsulates unique ids for HTML elements. The API is composed of an abstract type, `id`, and two operations. The server function `with_id` takes an HTML element, generates a fresh id and returns a pair composed of the HTML element with that id and the id. The client function `find` takes an id and retrieves the associated element as a DOM node on the client. The `id` type is abstract. Both the client and the server functions can use the real definition of `id` since they are both inside the module. The outer world, however, can not. Mixed modules allow us to allow abstraction to extends over the client-server boundary. This can provide further benefits in the case of more complex data-structures, as we will see in the next section.

```
module%mixed HtmlID : sig
  type id
  val%server with_id : Html.t -> Html.t * id
  val%client get : id -> DomHtml.t
end
```

Figure 5: Interface of abstract HTML ids

```
module%mixed HtmlID = struct
  type id = string
  let%server with_id elem =
    let myid = random_string () in
    let elem_with_id = Html.add_id elem myid in
    (elem_with_id, myid)
  let%client find myid = get_element_by_id myid
end
```

Figure 6: Implementation of abstract HTML ids

4.4 Mixed data structures

We now want to implement a system of client-side search and filtering of comments. The user should be able to search and filter comments directly on the client, without the need to reload the page. For this purpose, we need to maintain the sets of comments both on the server and on the client. One simple way to do that is to create a replicated cache of comments which ensures that all the comments available on the server are also available on the client.

We use the `Map` module as inspiration and create a functor that takes as argument a module describing the keys. 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.

The result API is shown in Fig. 7. The resulting module contains both a client and a server side types, both named `'a table`, which represent the local table. The module also exposes traditional `Map` functions. The implementation, shown in Fig. 8, is more interesting. We exploit the fact that client and server namespaces are distinct, and name both client and server map modules `M`. On the server, the cache is implemented as a pair of a server-side and a client-side dictionary. The server-side `add` implementation stores a new value locally in the expected way, but additionally builds a fragment that has the side-effect of performing a client-side addition. The retrieval operation (`find`) returns a shared value that contains both the server side version and the client side. On the client, however, we can directly use the local values. Since the client-side type exactly corresponds to a regular map, we can directly use the usual definitions for the various map operations. This is done by including the client `M` module on the client.

Note that this functor cannot be implemented in a decomposed way without sacrifying either abstraction or modularity. Indeed, the server implementation relies on the client-side version of the functor argument (`Comparable`) to implement proper usage of the keys. Furthermore, the signature of the functor ensures that the server-side and client-side parts of the cache are in sync without leaking any implementation details. Separating this mixed functors in two would require exposing the guts of the data-structure. Abstraction also makes it easy to extend such modules with new features. For example, it would possible to add full-blown replication through "push" or "pull" communications between the client and the server. Thanks to the abstraction provided by the signature of the module, this can even be done while keeping the API of the functor unchanged.

We can now use this cache for our comment system by using, for example, the `DateID` module for the keys. This is done in Fig. 9. Adding a new comment to the page is done through the `add_comment` server function. This function creates the associated HTML using the widget defined in Section 3.4 and adds it to the cache. We can then create the webpage containing all the comments simply by collecting all the comments and putting them inside a `div`. This is done by the `generate_page` server function. Finally, the client function `filter_comments` filters the shown comments on the client. It takes as argument a predicate function and the current client cache. It uses this predicate function to filter the cache, using the function `CommentCache.filter`, which directly uses the equivalent function from the `Map` module. We then find the HTML element containing all the elements and replace them them by the updated list.

Through these various examples, we demonstrated how we can combine traditional tierless features with advanced features of the OCaml module system to create powerful and expressive APIs. One one hand, tierless languages traditionally allows for complex interplay of client and server code. Module systems, on the other hand, allows to manipulate large pieces of code while preserving abstraction, encapsulation and modularity. The Eliom module system, and mixed functors in particular, allows to preserve these abstraction capabilities while enjoying the free-form tierless programming style.

```
module%mixed MakeCache (Key : COMPARABLE) : sig
  type%client 'a t
  type%server 'a t

  val%client add : Key.t -> 'a -> 'a t -> 'a t
  val%server add : Key.t -> 'a -> 'a t -> 'a t
  (* ... *)
end
```

Figure 7: Interface of `MakeCache`

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

  include%client M

  type%server 'a table = 'a M.t * 'a M.t fragment
  let%server add id v (tbl_server, tbl_client) =
    [%client M.add ~%id ~%v ~%tbl_client ];
    M.add id v tbl_server
  (* ... *)
end
```

Figure 8: Implementation of `MakeCache`

```
module%mixed DateKey = DateID
module%mixed CommentCache = MakeCache(DateKey)

let%server add_comment id cache =
  let html = make_comment id in
  CommentCache.add id html cache

let%server generate_page cache =
  Html.div a:[a_id "comments"] [CommentCache.elements cache]

let%client filter_comments predicate cache =
  let filtered_cache = CommentCache.filter predicate cache in
  let comment_container = get_element_by_id "comments" in
  Dom.replace_children
   comment_container
   (CommentCache.elements filtered_cache)
```

Figure 9: Using `MakeCache`

5 UNDER THE HOOD

Rich module systems such as ML's are notoriously difficult to formalize and implement. The best evidence of this is the very rich body of work attempting to provide a theoretic background for modules [8, 9, 15–17, 31, 36] compared to the very few implementations in modern languages [18, 21]. Adding tierless elements to the mix certainly does not make the situation simpler.

For space reasons, we do not attempt to provide a complete description of our module language. Instead, we highlight a few key elements that are novel in our approach. We first present some notes in the typechecking of modules (Section 5.1) and the compilation process (Section 5.3).

5.1 Typechecking and Specialization

On several occasions, we used base, client, server or even mixed modules in conjunction. We even applied a base functor such as `Map.Make` on a client module. Typechecking such a mix of base and non-base modules is not so trivial. Indeed, let us consider the functor application `Map.Make(JsDate)` presented in Section 4.2. Both the input and output signatures of `Map.Make` contain base fields. However, `JsDate` only contains client fields. Furthermore, one would expect `JsMap`, the result of the application, to be only usable on the client. In all these cases, we must "specialize" the `Map.Make` module to be usable on the client. This problem is similar to the application of a polymorphic function. Indeed, when checking the application of a function of type $\forall\alpha.\alpha \to \alpha$ to an argument of type *int*, we first instantiate the function to $int \to int$ before checking the application.

We use a similar technique to typecheck tierless modules. Instead of a set of type variables, we have a single "location variable" that is always called "base". When using a module in a client or a server context, we specialize it to ensure that all the fields are properly accessible. The specialization operation, noted $\lfloor M \rfloor_\ell$ where ℓ is "client" or "server", projects a "view" of the type of the module where all the fields are in the current location. For base modules, it simply rewrites the signature by substituting all instances of the location "base" by the specified "client" or "server" location. Fig. 10 presents two example of specialization for base modules. Note that before being specialized, a module should be actually accessible in the given scope. This means that we never have to specialize a server module on the client (or conversely).

The important part however is that specialization is completely transparent for the user. Much like instantiation of polymorphic function, specialization is automatically handled by the typechecker and requires no special care from the programmer. The programmer only has to specify client, server and base locations.

```
sig                                  sig
  type%base t                          type%client t
  val%base x : t          ⟶           val%client x : t
end                                  end
functor(M : S)T           ⟶         functor(M : ⌊S⌋)⌊T⌋
```

Figure 10: Examples of specialization – $\lfloor M \rfloor$

5.2 Mixed modules

Specialization is also used to enforce proper location usage for mixed modules. Indeed, mixed modules can be used on the client and on the server. In these cases, only the server (resp. client) part should be visible.

Mixed structures. For a structure, aka a collection of declarations, specialization hides the parts of a module that are not relevant to the current side. An example of specialization of a mixed structure is provided in the top half of Fig. 13. Here we can see that, as is the case for base modules, base declarations are now client. Furthermore, we also remove all the server declarations present in the structure. The end result is a structure that only contains client declarations. From a runtime point of view, the specialized type is also faithful to the content of the module: indeed, a base declaration can always be considered to be present client-side (as well as server-side) and declarations can be hidden thanks to module subtyping. This way, we ensure that if a structure (`struct` M `end`) can be given a type (`sig` S `end`), then it can also be given a type $\lfloor$`sig` S `end`$\rfloor_\ell$.

Mixed functors. Functors bring additional complexity. A naive implementation of specialization of mixed functors would be to specialize on both side of the arrow and apply the resulting functor. Let us see on an example why this solution does not work. In Fig. 11, the functor `F` takes as argument a module containing a base declaration and uses it on both sides. If the type of the functor parameter were specialized, the functor application in Fig. 12 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'`.

The solution here is that specialization on mixed functors should only specialize the return type, not the argument type. This is demonstrated in the bottom half of Fig. 13. This way, the complete mixed module is given as argument to the mixed functor and specialization happens on the result of the functor only.

5.3 Compilation and execution

In Section 3.5, we presented the semantics in term of a two-stage execution: first the server, then the client. This interpreted semantics is easier to understand, but would involve runtime code generation which would be quite inefficient. In the implementation, the Eliom compilers slices tierless programs in two parts, the client and the server. The slicing is done by emitting two OCaml programs containing additional communication primitives.

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

Figure 11: A mixed functor using a base declaration

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

Figure 12: An ill-typed application of F

```
sig                                    sig
  type%base t                            type%client t
  val%client x : int        ⟶           val%client x : int
  val%server y : t                     end
end
functor_mixed(M : S)T        ⟶   functor_mixed(M : S)⌊T⌋
```

Figure 13: Examples of specialization on mixed modules

For one-sided modules, the process is fairly simple: we simply take the whole module to the appropriate side. Similarly for mixed structures, we cut the whole structure in two. Complications arise for mixed functors. The idea is the following: we equip each mixed module with a unique identifier. This identifier is static for mixed structures but is dynamic for modules resulting of a functor application, such as F. On the server, this identifier is added as a field of the module. On the client, we simply maintain a table from identifiers to modules. When applying a functor, we remember the fact that the associated functor application should be done on the client. When sending the page to the client, we also send this information. The client will then ensure that this functor application is done at the appropriate time. This process can be seen as an extended version of the one used for Eliom fragments [30].

This compilation method also hints at some limitations of mixed functors: arguments of mixed functors must have an identifier pointing to their client half. One method is to add these identifiers to every mixed structures and force arguments of mixed functors to also be mixed. This restriction can be partially lifted through some simple static analysis to insert identifiers appropriately. Evaluating how constraining these restriction are in practice is the subject of future work.

5.4 Formal description

Radanne [28] gives a formal description of both the expression and the module language. It presents the type system, the module system, the interpreted semantics presented in Section 3.5 and the compilation scheme. This formalization demonstrates, among other things that the integration with OCaml works and that interpreted and compiled semantics coincide.

6 RELATED WORK

A comprehensive comparison of the tierless expression language can be found in Radanne et al. [29]. It is notoriously delicate to compare modules systems. Instead, we focus on the modularity and abstraction aspects and in particular the interaction between tierless programming, separate compilation and data abstraction. Within these criteria, the various approaches can be separated into three categories: slicing as a global compiler transformation, interpreted languages and modular compiled languages.

6.1 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 relies on some other program transformations (inlining, monomorphisation, defunctorisation, ...) that tend to be non-modular and cross abstraction boundaries.

Ur/Web [5, 6] is a statically typed ML-like tierless programming language. It only provides compilation units, not modules. Its approach to compilation is similar to MLton [23]: it applies a set of whole-program optimizations to remove all high order calls, then slices the program. This process is incompatible with separate compilation.

There has been a lot of work on bringing static slicing to JavaScript [26, 27]. 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.

6.2 Dynamic slicing

Some interpreted languages relies 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 [3, 34] is a dialect of Scheme for programming Web applications. Its successor, Hop.js [33] 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 [20] is a framework where both client and server side of an application are written in JavaScript. It does not provide static typing nor any form of abstraction.

Links [7] 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 [4], but it relies on normalization by evaluation, which is not immediately compatible with separate compilation.

6.3 Modular languages

Haste [10] 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. Kilpatrick et al. [13] developed a complete expressive module language for Haskell.

MetaOCaml [14] is an extension of OCaml for staged metaprogramming. While the expression language is quite similar to 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 [24, 39] 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 and the slicing can also be seen as dynamic (since code is executed at compile time). In particular, this allows to lift most of the restriction imposed on multi-stage functors.

Acute [35] 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 message and by versioning APIs.

CONCLUSION

We presented a module system for Eliom, a statically typed functional tierless Web programming language based on OCaml. It combines a powerful tierless expression language, as described by [29, 30], and a rich ML-style module language. To achieve this, we propose to annotate declarations with four *locations*: base, client, server or mixed. These locations allows to express tierless libraries conveniently while providing all the essential properties of a module system. In particular, Eliom is the only language that supports an efficient static compilation scheme, proper data abstraction and separate compilation. Furthermore, Eliom integrates seamlessly with the OCaml language and its ecosystem. We implemented this language as an extension of the OCaml compiler which includes typechecking, compilation and a runtime.

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.

REFERENCES

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

[2] 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.

[3] 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.

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

[5] Adam Chlipala. [n. d.]. Ur/Web: A Simple Model for Programming the Web *(POPL '15)*. https://doi.org/10.1145/2676726.2677004

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

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

[8] Karl Crary. 2017. Modules, abstraction, and parametric polymorphism, Giuseppe Castagna and Andrew D. Gordon (Eds.). http://dl.acm.org/citation.cfm?id=3009892

[9] Derek Dreyer. 2005. *Understanding and Evolving the ML Module System*. Ph.D. Dissertation. CMU. https://people.mpi-sws.org/~dreyer/thesis/main.pdf

[10] Anton Ekblad and Koen Claessen. [n. d.]. A Seamless, Client-centric Programming Model for Type Safe Web Applications *(Haskell '14)*. https://doi.org/10.1145/2633357.2633367

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

[12] Hashids 2017. *Hashids*. http://hashids.org/.

[13] Scott Kilpatrick, Derek Dreyer, Simon L. Peyton Jones, and Simon Marlow. 2014. Backpack: retrofitting Haskell with interfaces. https://doi.org/10.1145/2535838.2535884

[14] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml *(Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.), Vol. 8475. Springer. https://doi.org/10.1007/978-3-319-07151-0_6

[15] Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a mechanized metatheory of standard ML, Martin Hofmann and Matthias Felleisen (Eds.). ACM. https://doi.org/10.1145/1190216.1190245

[16] Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press. https://doi.org/10.1145/174675.176926

[17] Xavier Leroy. 1995. Applicative Functors and Fully Transparent Higher-Order Modules, Ron K. Cytron and Peter Lee (Eds.). ACM Press. https://doi.org/10.1145/199448.199476

[18] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2016. *The OCaml system release 4.04, Documentation and user's manual*. Projet Gallium, INRIA.

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

[20] Meteor.js 2017. *Meteor.js*. http://meteor.com.

[21] Robin Milner, Mads Tofte, and Robert Harper. 1990. *Definition of standard ML*. MIT Press.

[22] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. 2013. *Real World OCaml - Functional Programming for the Masses*. O'Reilly. https://realworldocaml.org/

[23] MLton 2014. MLton. (2014). http://mlton.org/Home

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

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

[26] Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. 2014. Towards Tierless Web Development Without Tierless Languages *(Onward! 2014)*. 69–81. https://doi.org/10.1145/2661136.2661146

[27] Laure Philips, Joeri De Koster, Wolfgang De Meuter, and Coen De Roover. 2016. Dependence-driven delimited CPS transformation for JavaScript. https://doi.org/10.1145/2993236.2993243

[28] Gabriel Radanne. 2017. *Tierless Web Programming in ML*. Ph.D. Dissertation. Paris Diderot. https://www.irif.fr/~gradanne/papers/phdthesis.pdf

[29] Gabriel Radanne, Vasilis Papavasileiou, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: tierless Web programming from the ground up, Tom Schrijvers (Ed.). ACM. https://doi.org/10.1145/3064899.3064901

[30] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2016. Eliom: A Core ML Language for Tierless Web Programming *(Lecture Notes in Computer Science)*. https://doi.org/10.1007/978-3-319-47958-3_20

[31] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 2014. F-ing modules. *J. Funct. Program.* 24, 5 (2014), 529–607. https://doi.org/10.1017/S0956796814000264

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

[33] Manuel Serrano and Vincent Prunet. 2016. A glimpse of Hopjs, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM. https://doi.org/10.1145/2951913.2951916

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

[35] 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.* (2007). https://doi.org/10.1017/S0956796807006442

[36] Mads Tofte. 1988. *Operational Semantics and Polymorphic Type Inference*. Ph.D. Dissertation. University of Edinburgh.

[37] TyXML 2017. *TyXML*. http://ocsigen.org/tyxml/.

[38] Jérôme Vouillon and Vincent Balat. 2014. From bytecode to JavaScript: the Js_of_ocaml compiler. *Software: Practice and Experience* (2014). https://doi.org/10.1002/spe.2187

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