Eliom: A Language for Modular Tierless Web Programming

Gabriel Radanne, Jérôme Vouillon, Vincent Balat

Eliom: A Language for Modular Tierless Web Programming

GABRIEL RADANNE, University of Freiburg, Germany
JÉRÔME VOUILLON, Univ Paris Diderot, Sorbonne Paris Cité, France, BeSport, France, and CNRS, France
VINCENT BALAT, Univ Paris Diderot, Sorbonne Paris Cité, France and BeSport, France

Tierless Web programming languages allow programmers to combine client-side and server-side programming in a single program. Programmers can then define components with both client and server parts and get flexible, efficient and typesafe client-server communications. However, the expressive client-server features found in most tierless languages are not necessarily compatible with functionalities found in many mainstream languages. In particular, we would like to benefit from type safety, an efficient execution, static compilation, modularity and separate compilation.

In this paper, we propose Eliom, an industrial-strength tierless functional Web programming language which extends OCaml with support for rich client/server interactions. It allows to build whole applications as a single distributed program, in which it is possible to define modular tierless libraries with both server and client behaviors and combine them effortlessly. Eliom is the only language that combines type-safe and efficient client/server communications with a static compilation model that supports separate compilation and modularity. It also supports excellent integration with OCaml, allowing to transparently leverage its ecosystem.

To achieve all these features, Eliom borrows ideas not only from distributed programming languages, but also from meta-programming and modern module systems. We present the design of Eliom, how it can be used in practice and its formalization; including its type system, semantics and compilation scheme. We show that this compilation scheme preserves typing and semantics, and that it supports separate compilation.

CCS Concepts: • **Software and its engineering → Functional languages**; **Modules / packages**; **Formal language definitions**; **Compilers**; *Distributed programming languages*;

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

ACM Reference Format:
Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. 2019. Eliom: A Language for Modular Tierless Web Programming. *ACM Trans. Program. Lang. Syst.* 1, 1 (January 2019), 87 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Writing websites nowadays requires the use of many different technologies: HTML and CSS for the content and visuals of the website, JavaScript for client interactions, any of the numerous server languages (PHP, Ruby, C#, …), a database language such as SQL, etc. Not only do programmers

This work was partially performed at IRILL, center for Free Software Research and Innovation in Paris, France, http://www.irill.org.
Authors' addresses: Gabriel Radanne, University of Freiburg, Germany, radanne@informatik.uni-freiburg.de; Jérôme Vouillon, IRIF UMR 8243 CNRS, Univ Paris Diderot, Sorbonne Paris Cité, Paris, France , BeSport, Paris, France , CNRS, France, jerome.vouillon@irif.fr; Vincent Balat, IRIF UMR 8243, Univ Paris Diderot, Sorbonne Paris Cité, Paris, France , BeSport, Paris, France, vincent.balat@irif.fr.

0164-0925/2019/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

need to use all these technologies, but they also need them to cooperate in an harmonious manner by ensuring that communications between these various components are correct. Such verification is often done manually by the programmer, which is both error-prone and time-consuming.

This juggling of many different technologies does not only make programming more complicated, it also imposes strong constraints on code organization and prevents modularity. Let us consider the case where we want to add a comment widget to a website. This widget requires JavaScript code for client interactions, for example a convenient editor. It also requires server code that will store and serve the comments and potentially some associated database queries. This code will thus be split in two codebases and up to three languages. Furthermore, in order to be properly integrated in the larger website, the internal communications between the client and server parts of the widget will be exposed to the rest of the program. All these programming constraints, which stem directly from the way Web programming is done currently, make it impossible to preserve abstraction and encapsulation for widgets who have both client and server aspects. This hinders safety and reusability of widgets by preventing programmers to create self-contained independent libraries that have both client and server aspects.

1.1 Ocsigen

Ocsigen provides a comprehensive set of tools and libraries for developing Web applications in OCaml. OCaml is an industrial strenght statically typed functional programming language with a rich ecosystem. The Ocsigen project includes the compiler js_of_ocaml [Vouillon and Balat 2014], a Web server, libraries for concurrency [Vouillon 2008] and HTML manipulation [TyXML 2017]. It also contains a complete framework to develop complex Web application with both client and server components. It includes modules for, among other things, RPCs, Web GUIs [Ocsigen Toolkit 2017], functional reactive Web programming and an advanced *service identification mechanism* [Balat 2014]. Ocsigen is already used in production for a wide variety of websites [Besport 2017; Gencore 2017; Pumgrana 2017].

Based on our experience building such a framework, we realized we needed additional tools to bridge the gap between client and server components. Indeed, while OCaml provides excellent support for abstraction and modularity through its module language, it doesn't allow to talk directly about the relationship between client and server code. In particular, we needed language constructs that allows us to blur the boundaries between client and server. Such language constructs can notably be provided by tierless languages.

1.2 Tierless Web Programming Languages

Tierless languages [Chlipala 2015b; Cooper et al. 2006; Serrano et al. 2006] allow programmers to write in a composable way programs that have both client and server aspects. They provide language constructs to define on the server functions that create fragments of a Web page together with their associated client-side behavior. This is done by allowing to freely intersperse client and server expressions with seamless communication in a unique programming language. By grouping together client and server parts, they allow to encapsulate hybrid libraries and create more modular websites. To be executed, tierless programs are sliced in two: a part which runs on the server and a part which is compiled to JavaScript and runs on the client. This slicing can be done either dynamically, by generating JavaScript code at runtime, or statically, by cutting the program in two during compilation. Tierless languages can also leverage static typing to statically ensure that communications between client and server are always correct. Such tierless languages can also leverage both functional programming and static typing. Functional programming provides increased expressiveness and flexibility while static typing gives strong guarantees about client-server separation, in particular ensuring that communications are consistent across tiers.

To complete the design of Ocsigen, we needed a language that not only provides such expressive tierless features, but also provides support for most features already supported by OCaml: type safety, efficiency, static compilation, modularity and separate compilation. Furthermore, Web programming often relies on numerous external libraries (for encryption, HTML, logging, ...). We wanted to leverage all the existing tools and libraries developed either in the Ocsigen project or the larger OCaml ecosystem.

1.3 The design of a Modular Tierless language

To build such a language with support for both modular and tierless features, we outline several goals that will guide the design of our language.

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

A simple and efficient execution model. The language should not incur additional back-and-forth communications between client and server. Furthermore, the execution model should be simple and predictable. Having a predictable execution model is essential in the context of a language which is not purely functional, like OCaml.

Leveraging the type system. The language should leverages the type system to allows composition and modularity of client-server programs while preserving type-safety and abstraction. This ensures, via the type-system, that client functions are not called by mistake inside server code (and conversely) and ensures the correctness of client-server communications.

Integration with the host language. Programmers must 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, should be a valid module in our language. Furthermore, we should be able to specify if we want to use a given library on the client, on the server, or everywhere.

Modularity and encapsulation. Module and type abstractions are very powerful programming tools. By only exposing part of a library, the programmer can safely hide implementation details and enforce specific properties. The language should 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 fine-grained details of client-server communication to the users.

1.4 Contributions and Plan

We propose Eliom, a tierless web programming language that supports static typing, an efficient compilation scheme that avoids extra communications and a very powerful form of modularity inspired by ML-style languages. In particular, Eliom is the only tierless programming language featuring efficient static separate compilation. Eliom extends OCaml and can transparently leverage its complete ecosystem. This article presents the design of Eliom, how it can be used in practice, and formalizes the underlying core language. Our contributions are the following:

- We designed Eliom, a tierless web programming language that follows the set of goals presented above. We demonstrate the practicality of our language through examples (Section 2).
- We formalized the type system and the semantics of Eliom, including both the expression and the module layers (Section 3). This formalization provides an *interpreted* semantics that is easy to reason with and can be explained to programmers, but still maintains Eliom's good properties regarding communications and typesafety. Using this semantics, we show that Eliom supports separate typechecking (Theorem 1) and complete integration with OCaml (Section 4 and Theorem 2).
- As highlighted before, Eliom is not an interpreted language. We formalized the compilation model of Eliom (Section 5.2). This compilation scheme turns one Eliom program into two simpler OCaml programs; one for the client and one for the server. We show that such compilation scheme supports separate compilation and preserves both the typing and the semantics of Eliom (Theorems 3 and 4).
- We implemented Eliom as a patch on the OCaml compiler and a runtime library[1].

2 PROGRAMMING WITH ELIOM

An Eliom application is composed of a single program that has both client and server behaviors. During compilation, the Eliom compiler decomposes the program 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 second part, compiled statically to JavaScript, is sent to each client by the Web server together with the HTML page, in response to the initial HTTP request.

In this section, we give a tour of the Eliom language through various examples. As a guiding example for our exploration of the Eliom language, we consider the case of a commenting system similar to websites such as Reddit or Hackernews, A comment is a piece of HTML written by a user and identified by a unique identifier. Comments are initially stored on the server. Users can then manipulate them on the client by voting, hiding or searching them. Such a library features both server aspects (storing and rendering the comments) and client interactions (browsing and searching comments). We start by giving a reminder of some important concepts on OCaml modules before presenting the various Eliom concepts. 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 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.

```
1 module On = struct
2   let click = ...
3   let keypress = ...
4 end
```

[1] https://github.com/ocsigen/ocaml-eliom and https://github.com/ocsigen/eliomlang

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.

2.1.1 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-based IDs or something else like UUIDs and Hashids [2017]. Fortunately, we do not have to make this decision immediately! All we need in order to write the rest of our library is an interface for creating and using identifiers. We can declare such an interface in OCaml using a *signature*. In Section 2.1.2, we declare the `ID` signature describing what a module implementing unique identifiers should look like. We then define two modules implementing this specification, `SequentialID` and `DateID`, which we can switch easily.

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 rest of this section.

2.1.2 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 Figure 3, we use this functor to create the

```
module type ID = sig
  type t (* type of ids *)

  val compare : t -> t -> int
  val create : unit -> t
  val to_string : t -> string
end
```

```
module SequentialID : ID = struct
  type t = int

  (* ... *)
end
```

```
module DateID : ID = struct
  type t = date

  (* ... *)
end
```

Fig. 1. The `ID` signature and two implementations.

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

```
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
```

Fig. 2. the `Map` module

```
module TheID = DateID (* The ID of our choice *)
module IDMap = Map.Make(TheID)

let register c map : Html.t IDMap.t =
  let commentid = TheID.create () in
  IDMap.add commentid c map
```

Fig. 3. Dictionaries from IDs to comments

2.2 Tierless widgets

Until now, we presented how to write various elements of libraries useful for our comment system by leveraging the power of the OCaml module system in various ways. We now want to write the widget that presents a comment. We thus need to define both client and server code, along with some client-server communication, which is precisely where tierless languages shine.

2.2.1 Declarations. Locations in Eliom are explicit. Each declaration must be marked with an annotation that specifies whether a declaration is to be performed on the server or on the client as follows:

```
let%server s = ...
type%client t = ...
```

Every declaration, such as types, values and modules, can be annotated. These annotations allows 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** and server is in **blue**. Colors are however not mandatory to understand the rest of this article.

2.2.2 Fragments. While location annotations allow programmers to gather code across locations, they don't allow convenient communication. For this purpose, Eliom allows to include client-side expressions inside server declarations: 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 ]
```

2.2.3 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
```

2.2.4 Comment widget. These three constructions are sufficient to create complex client-server interactions such as the comment widget. The comment widget shows a single comment and can be used several times to show a list of comments. It not only shows the author and the content of the comment, but will also hide the content when the user clicks on it. Finally, we want the HTML to be generated server-side and sent to the client as a regular HTML page, which allows the comments to be accessible even when JavaScript cannot run. The implementation, the interface and the produced HTML fragment are shown in Figure 4.

In order to implement our comment widget, we use an HTML DSL [TyXML 2017] that provides combinators such as `div` and `a_onclick` (which respectively create an HTML tag and an HTML attribute). In OCaml, `~a` denotes a named argument that is used here to provide the list of HTML attributes. We first create a `p` element which contains 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.

The signature of our widget function, shown Figure 4b, 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.

2.2.5 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. The client code is executed only once the Web page has been sent to the client. We also guarantee that client code, be it either client declarations or fragments, is executed in the order that it was encountered on the server.

```
open Html
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]
```

(a) Implementation

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

(b) Interface

```
<div>
  Author: The Ocsigen Team
  <p id=42>I'm a composable widget!</p>
</div>
```

(c) Resulting HTML

Fig. 4. The comment widget

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 preserves 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.

2.3 Hybrid data-structures

We want to add some buttons to our comment widget for various possible actions on comments such as "reply", "permalink", "report", etc. We want our buttons to be created in a uniform manner as a list of symbolic actions. Some of our actions are normal links and some are client-side actions. Thanks to fragments, we can create hybrid data-structures that have both client and server parts. Figure 5 implements such an hybrid data-structure to specify button actions. The `action` type is a server type that is either a normal link or a client-side action (here, a function from unit to bool) in a fragment. The `attr_of_action` function turns actions into HTML attributes, either a link or an on-click handler. Finally, `button_list` walks through a list of pairs of names and actions and returns an unordered list of `a` elements.

Creating such hybrid data-structures allows a great flexibility in how Eliom code is organized by allowing to meld client and server code at any point in the application. Having explicit annotations for client and server code of is essential here. This would be quite difficult to achieve if the delimitation between client and server values were implicitly inferred.

2.3.1 Dynamic loading of comments. In order to handle pages with many comments, we want to only present a subset of the comment upfront and dynamically load additional comments when the user asks for it (or, alternatively, when the page scrolls). This requires dynamic communications which are not directly provided by fragments and injections, as indicated in the previous section. For such purposes, Eliom provides the `Rpc` module whose API is presented in Figure 6.

We implement a button which dynamically loads new comments in Figure 7. We first create server-side an RPC endpoint `fetch_comments` with the function `Rpc.create`. This endpoints is of type `(page_id, Html.t list)Rpc.t`: it takes as argument the id of the current page and return the list of associated comments using the pre-defined function `Comments.from_page`. The type `Rpc.t` is abstract on the server, but is a synonym for a function type on the client. Of course,

```
type%server action =
  | Link of Url.t
  | Action of (unit -> bool) fragment

let%server attr_of_action action = match action with
  | Link l -> a_href l
  | Action f -> On.click f

let%server button_list (lst : (name * action) list) =
  ul (List.map (fun (name, action) ->
      li [a ~a:[attr_of_action action] [pcdata name]])
      lst)
```

Fig. 5. Function generating a list of buttons

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

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

Fig. 6. Rpc signature

```
let%server fetch_comments
  : (page_id, Html.t list) Rpc.t
  = Rpc.create Comment.from_page

let%client button_load parent page_id =
  let handler _ =
    let comments = ~%fetch_comments page_id in
    List.iter Dom.append_child parent comments
  in
  Html.button
    ~a:[On.click handler]
    [text "Load more comments"]
```

Fig. 7. Dynamic loading of comments with Rpc

this function does not contain the actual implementation of the RPC handler, which only exists server-side. To use this API, we leverage injections. By using an injection on `~%fetch_comments` on Line 7, we obtain *on the client* a value of type `Rpc.t`. We describe the underlying machinery that we leverage for converting RPC endpoints into client-side functions in Section 2.3.2. What matters here is that we end up with a function that we can call like any other; calling it performs the remote procedure call. Once we have a way to fetch new comments, we can create a button `button_load` which, when clicked, loads the comments. This is done by creating a client-side handler and using the `On.click` function.

The RPC API proposed in Figure 6 is "blocking": the execution waits for the remote call to finish before pursuing, thus blocking the rest of the client program. Remote procedure calls should be made asynchronously: the client program keeps running while the call is made and the result is used when the communication is done. The actual implementation uses Lwt [Vouillon 2008] to express asynchronous calls in a programmer-friendly manner through promises. The use of Lwt is pervasive in the Eliom ecosystem both on the server and on the client. In this article, we omit mentions of the Lwt types and operators for pedagogic purposes.

2.3.2 Converters. In the RPC API, we associate two types with different implementation on the server and on the client. We rely on injections to transform the datastructure when moving from one side to the other. This ability to transform data before sending it to the client via an injection is made possible by the use of *converters*. Figure 8 broadly presents the converter API. Given a serialization format `serial`, a converter is a pair of a *server* serialization function and a *client* de-serialization function. Note that the client and server types are not necessarily the

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

Fig. 8. Schematized API for converters

```
type%server ('i,'o) t = {
  url : string ;
  handler: 'i -> 'o ;
}
type%client ('i, 'o) t = 'i -> 'o

let%server serialize t = serialize_string t.url
let%client deserialize x =
  let url = deserialize_string x in (fun i -> XmlHttpRequest.get url i)

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

let%server create handler =
  let url = "/rpc/" ^ generate_new_id () in
  serve url handler ;
  { url ; handler }
```

Fig. 9. Simplified RPC implementation corresponding to Figure 6.

same. Furthermore, we can arbitrarily manipulate the value before returning it. Several predefined converters are available for fragments, basic OCaml datatypes, and tuples in the module `Conv`. Implementation details about converters can be found in Section 3.3.2.

We can use converters to implement the RPC API (Figure 9). The server implementation of `Rpc.t` is composed of a handler, which is a server function, and a URL to which the endpoint answers. Our serialization function only sends the URL of the endpoint. The client de-serialization function uses this URL to create a function performing an HTTP request to the endpoint. This way, an RPC endpoint can be accessed simply with an injection. Thus, for the `create` function, we assume that we have a function `serve` of type `string -> (request -> answer)-> unit` that creates an HTTP handler at a specified URL. When `Rpc.create` is called, a unique identifier `id` is created, along with a new HTTP endpoint `"/rpc/id"` that invokes the specified function.

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

2.4 Modular Tierless 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 2.1, which provides abstraction and modularity at the library level. On the other hand, we have a powerful tierless programming language, as presented in Section 2.2, 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.

2.4.1 Interaction with OCaml. In the previous examples, we freely used OCaml modules in server and client contexts. In Figure 8, we even defined an API that has both normal OCaml declarations and server declarations! 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. The ability to transparently leverage the OCaml ecosystem in Eliom is essential to write web applications productively.

This close integration 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.

2.4.2 Modules and locations. Since OCaml modules, such as the `Html` module defined earlier, 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 'a t

  val empty : 'a t
  val add : Js.string -> '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 2.1.2. 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 Figure 10. 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

  (** Compare by timestamp *)
  let compare x y = compare x##valueOf y##valueOf
end

module%client JsDateMap = Map.Make(JsDate)
```

Fig. 10. Definition of `JsDate` and `JsDateMap`

2.4.3 Abstraction and encapsulation across locations. 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 2.2.4. 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 directly, as a string, 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.

One solution, which we used in Section 2.3.2, is to use converters and only use the ID as an implementation detail of the client-server communication. However, it is sometimes beneficial to keep an explicit ID. By combining tierless annotations and the abstraction capabilities provided by modules, we can have explicit ID that are abstract and type-safe. Figure 11 presents an API that encapsulates unique ids for HTML elements. This new `HtmlID` is **mixed** and can contain client, server and base declarations. It is composed of an OCaml 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 itself. 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 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.

2.4.4 Multi-tiers comments. 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

```
module%mixed HtmlID : sig
  type id

  val%server with_id : Html.t -> Html.t * id

  val%client get : id -> DomHtml.t
end
```

(a) Interface

```
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
```

(b) Implementation

Fig. 11. Abstract HTML ids for client/server communications

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 Figure 12. 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 Figure 13, 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 Figure 14. 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 2.2.4 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 by the updated list. Since the `CommentCache` module also contains a *client-side* add function, this approach works very well the dynamic loading of comment presented in Section 2.3.1: Once the RPC returns the new comments, we can add them to the client-side cache directly.

```
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
```

Fig. 12. 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
```

Fig. 13. 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)
```

Fig. 14. Using MakeCache

A note on mixed functors. Mixed functors unfortunately have some limitations. Arguments must themselves be mixed modules and injections inside client-side bindings can only reference elements outside of the functor. Additionally, there are some limitations regarding nesting of mixed structures and functors. These limitations are formalized and discussed more precisely in Section 3.4.2. Mixed functors are nevertheless sufficient for a large class of complex client-server APIs that are useful in practice, as demonstrated with the MakeCache functor.

2.5 Going further

Through these various examples, we demonstrated how we can combine traditional tierless features with an advanced 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. Eliom allows to preserve these abstraction capabilities while enjoying the free-form tierless programming style. The Ocsigen ecosystem uses these concepts to provide numerous additional tools such as advanced communication patterns, shared HTML across tiers or even distributed Functional Reactive Programming [Radanne et al. 2016a; Radanne and Vouillon 2018; Tutorial 2017]. All these advanced mechanisms are used in production by Eliom users.

3 THE ELIOM$_\varepsilon$ CALCULUS

We now formalize Eliom as an extension of ML with both an expression and a module language. To emphasize the new elements introduced by Eliom, these additional elements will be colored in blue. This is only for ease of reading and is not essential for understanding the formalization.

While Eliom is an extension of OCaml, Eliom$_\varepsilon$ is an extension of a simpler ML calculus, which we present quickly in Section 3.1. We then present the location system in Section 3.2, followed by the expression and the module languages in Sections 3.3 and 3.4. Finally, we describe its semantics in Section 3.6.

Syntactic considerations. Let us first define some notations and meta-syntactic variables. As a general rule, the expression language is in lowercase (e) and the module language is in uppercase (M). Module types are in calligraphic letters ($\mathcal{M}$). More precisely: e are expressions, x are variables, p are module paths, X are module variables, τ are type expressions and t are type constructors. x_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 (following Leroy [1995]): α-conversion should keep the name intact and change only the stamp. Sequences are noted with a star; for example τ^* is a sequence of type expressions. Indexed sequences are noted (τ_i), with an implicit range. Substitution of a by b in e is noted $e[a \mapsto b]$. Repeated substitution of each a_i by the corresponding b_i is noted $e[a_i \mapsto b_i]_i$.

3.1 A crash course in the ML calculus

There are many variants of the ML calculus. For the purpose of ELIOM$_\varepsilon$, we consider a core calculus with polymorphism, let bindings and parametrized datatypes in the style of Wright and Felleisen [1994], accompanied by a fully featured module system with separate compilation and applicative functors in the style of Leroy [1994, 1995]. The goal of this calculus is to closely module the most relevant features of OCAML, while being sufficiently simple to be extended and reasoned on formally.

In this section, we will simple give a quick overview of the syntax and an informal reminder of the most distinctive or unusual points. The complete treatment of the language is given in Appendix A. Each set of typing or reduction rules can also be obtained by considering the ELIOM$_\varepsilon$ rules in the next section and ignoring the parts written in blue.

3.1.1 Syntax. The complete syntax is presented in Figure 42 and follows the syntax of OCAML closely. The expression language is a fairly simple extension of the lambda calculus with a fixpoint combinator (Y) and let bindings ($\texttt{let}\ x = e_1\ \texttt{in}\ e_2$). The language is parametrized by a set of constants *Const*. Variables can be qualified by a module path p. Paths can be either module identifiers such as X_i, a submodule access such as $X_i.Y$, or a path application such as $X_i(Y_j.Z)$. Note that, as said earlier, that fields of modules are only called by their name, without stamp.

The module language is composed of functors, module application ($M_1(M_2)$), type constraints ($(M : \mathcal{M})$). and structures containing a list of value, types or module definitions ($\texttt{struct let}\ x_i = 2\ \texttt{end}$). Programs are lists of definitions. Module types can be either functor types ($\texttt{functor}(X_i : \mathcal{M}_1)\mathcal{M}_2$) or a signature ($\texttt{sig val}\ x_i\ \texttt{: int end}$), which contains a list of value, types and module descriptions. Type descriptions can expose their definition or can be left abstract. Typing environments are simply module signatures. We note them Γ for convenience.

3.1.2 Type system. As a general rule, judgements are denoted with the symbol $\triangleright$ for expressions and $\blacktriangleright$ for modules. More precisely:

$\Gamma \triangleright e : \tau$	Expression e has type τ in Γ
$\Gamma \triangleright \tau_1 \approx \tau_2$	τ_1 and τ_2 are equivalent in Γ
$\Gamma \blacktriangleright M : \mathcal{M}$	module M has type $\mathcal{M}$ in Γ
$\Gamma \blacktriangleright \mathcal{M} <: \mathcal{M}'$	Module type $\mathcal{M}$ is a subtype of $\mathcal{M}'$ in Γ
$\Gamma \vDash \tau$ and $\Gamma \vDash \mathcal{M}$	The type τ (resp. module type $\mathcal{M}$) is well formed.

The type system for expression is the standard Hindley-Milner type system with polymorphism, generalization and parametrized datatype. The module system, which follows Leroy [1994, 1995], is less usual. We will focus on three specific features: qualified accesses, applicative functors and strengthening.

Qualified accesses. Let us consider the module X with the module type (`sig type` t; `val` $a : t$ `end`). X contains a type t and a value a of that type. We wish to typecheck $X.a$. One expected type for this expression is $X.t$. However, the binding of v in X gives the type t, with no mention of X. In order to make the type of $X.a$ expressible in the current scope, we prefix the type variable t by the access path X. This is done in the rule QualModVar by the substitution $\mathcal{M}[n_i \mapsto p.n \mid n_i \in \mathrm{BV}(\mathcal{S}_1)]$ which prefixes all the bound variables of $\mathcal{S}_1$, noted $\mathrm{BV}(\mathcal{S}_1)$, by the path p. We thus obtain the following type derivation

$$\textsc{QualVar}\ \dfrac{\textsc{ModVar}\ \dfrac{(\texttt{module}\ X : \ldots) \in (\texttt{module}\ X : \ldots)}{(\texttt{module}\ X : \ldots) \blacktriangleright X : \texttt{sig type}\ t;\ \texttt{val}\ a : t\ \texttt{end}}}{(\texttt{module}\ X : \ldots) \triangleright X.a : \mathrm{X.t}}\ \text{with } X.t = t[t \mapsto p.t]$$

Applicative functors. In Section 2.1, we used the `Map.Make` functor to create a new dictionary data-structure using the provided type as keys. One might wonder what happens if `Map.Make` is applied to the same module twice. Are the dictionaries thus produced compatible? More formally, Does $M_1 = M_2$ implies $F(M_1) = F(M_2)$? When this holds, we say that functors are "applicatives". Otherwise, we say they are "generatives". OCaml functors are applicatives by default. In our simple ML calculus, all functors are applicatives. This is implemented by enriching type constructors with types of the form $F(M).t$. If we consider the modules $N_i = \mathrm{Map.Make}(M_i)$, then we have $N_1.t = \mathrm{Map.Make}(M_1).t = \mathrm{Map.Make}(M_2).t = N_2.t$.

Strengthening. Let us consider a module X answering the signature shown Example 20a. We want to pass this module to the functor F shown in Example 20b. This functor application is expected to succeed, since $X.t = t$, where t comes from X. However, the definition of t in X is abstract, and does not have a definition allowing us to check that X is indeed compatible with the proposed signature. For this purpose, we *strengthen* the type of the module X with additional type equalities. If $\mathcal{X}$ is the type of X, we note $\mathcal{X}/X$ the strengthened signature. The result is the signature (`sig type` $t = X.t$; `val` $v : t$ `end`). This signature is now trivially included in the argument type of F.

```
module X : sig
  type t
  val v : t
end
```

(a) Typing environment

```
module F (E : sig type t = X.t val v : t end) = struct
  ...
end
module X = F(X)
```

(b) Application of multi-argument functor using manifests

Fig. 15. Program using functors and manifest types

3.1.3 Semantics. We use a rule-based big-step semantics with traces. Big-step semantics are more amenable to reasoning with modules, as they allow to easily introduced local bindings. We record traces in order to reason about execution order. We note v for values in the expression language and V for values in the module language. Values in the expression language can be either constants or lambdas. Module values are either structures, which are list of bindings of values, or functors. Execution environments, noted ρ, are a list of value bindings. We note the concatenation of environment $+$. Environment access is noted $\rho(x) = v$ where x has value v in ρ. The same notation is also used for structures. Traces are lists of messages. The empty trace is noted $\langle\rangle$. Concatenation of traces is noted @.

$$\ell ::= s \mid c \mid b \qquad\qquad \varsigma ::= m \mid \ell$$

Fig. 16. Grammar of locations – ℓ and ς

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

Fig. 17. "can be used in" relations on locations – $\varsigma \succ \varsigma'$

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

Fig. 18. "can contain" relation on locations – $\varsigma <: \varsigma'$

Given an expression e (resp. a module M), an execution environment ρ, a value v (resp. V) and a trace θ,

$$e \overset{\rho}{\Longrightarrow} v, \theta$$

means that e reduces to v in ρ and prints θ. The rules are composed of traditional call-by-value lambda-calculus with a fixed execution order that is represented by the traces.

3.1.4 Formal results. The ML calculus considered here has been shown to support for separate compilation (Theorem 5) and representation independence [Leroy 1994]. It doesn't have a proof of soundness, which discussed in Appendix A.3.3.

3.2 Locations

Before introducing the Eliom$_\varepsilon$ language constructs, we consider the notation for *locations* annotations which indicate "where the code runs". The grammar of locations is given in Figure 16. There are three core locations: **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. There is a fourth location that is only available for modules: **m**ixed. A mixed module can have client, server and base components. We use the meta-variable ς for locations that are either **m** or one of the core locations. In most contexts, locations are annotated with subscripts.

We also introduce two relations:

$\varsigma \succ \varsigma'$ defined in Figure 17, means that a variable (either values, types or modules) defined on location ς can be used on a location ς'. For example, a base type can be used in a client context. Base declarations are usable everywhere. Mixed declarations are not usable in base code.

$\varsigma <: \varsigma'$ defined in Figure 18, means that a module defined on location ς can contain component on location ς'. In particular, the mixed location m can contain any component, while other location can contain only component declared on the same location.

Both relations are reflexive: For instance, it is always possible to use client declarations when you are on the client.

3.3 Expressions

Eliom$_\varepsilon$'s expression language is based on a simple ML language, as presented in Section 3.1, extended with Eliom specific constructs: fragments and injections. This expression language can be seen as an alternative formulation of the core language proposed by Radanne et al. [2016b], but that is compatible with modules.

A *client fragment* {{ e }} can be used on the server to represent an expression that will be computed on the client, but whose future value can be referred to 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 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. For clarity, we sometime 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* $\{\tau\}$ 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. All type variables α_ℓ are annotated with a core location ℓ. There are now three sets of constants: client, server and base.

3.3.1 Typing rules. Typing judgements for Eliom$_\varepsilon$ are annotated with a location that specifies where the given code should be typechecked. For the expression language, we only consider core locations ℓ which are either base, client or server. The typing judgment, defined in Figure 20, is noted $\Gamma \triangleright_\ell e : \tau$ where e is of type τ in the environment Γ on the location ℓ. We also define type validity and equivalence judgements in Figures 21 and 22. We note $TypeOf_\ell(c)$ the type of a given constant c on the location ℓ. Binding in typing environments, just like in signatures, are annotated with a location. 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 m. Names are namespaced by locations, which means it is valid to have different client and server values with the same name.

Most of the rules are straightforward adaptions of traditional ML rules and are described in Section 3.1. We focus on rules that demonstrate some particular features of Eliom$_\varepsilon$.

Rule Var contains a significant difference compared to the traditional ML rules. As described earlier, bindings in Eliom$_\varepsilon$ are located. Since access across sides are explicit, we want to prevent erroneous cross-location accesses. For example, client variables can not be used on the server. To use a variable x bound on location ℓ' in a context ℓ, we check that location ℓ' can be used in location ℓ, noted $\ell' \succ \ell$ and defined in Figure 17. Base elements b are usable everywhere. Mixed elements m are usable in both client and server.

Type variables and types constructors are also annotated with a location and follow the same rules. Using type variables from the client on the server, for example, is disallowed. Such constraints are enforced by the type validity and equivalence judgements though rules such as FragVal and ConvVal which tracks locations across type expressions.

Expressions

$e ::= c$	(Constant)
$\mid x_i \mid p.x$	(Variables)
$\mid \mathsf{Y}$	(Fixpoint)
$\mid (e\ e)$	(Application)
$\mid \lambda x.e$	(Function)
$\mid \texttt{let}\ x = e\ \texttt{in}\ e$	(Let binding)
$\mid \{\{\ e\ \}\}$	(Fragment)
$\mid f\%x$	(Injection)
$f ::= p.v \mid v_i \mid c$	(Converter)
$c \in Const$	(Constants)

Path

$p ::= X_i \mid p.X \mid p_1(p_2)$

Type Schemes

$\sigma ::= \forall(\alpha_\ell)^*.\tau$

Type Expressions

$\tau ::= \alpha_\ell$	(Type variables)
$\mid \tau \to \tau$	(Function types)
$\mid (\tau^*)\mathrm{t}_\mathrm{i} \mid (\tau^*)\mathrm{p.t}$	(Type constructors)
$\mid \{\tau\}$	(Fragment)
$\mid \tau \rightsquigarrow \tau$	(Converter types)

Fig. 19. Grammar for Eliom$_\varepsilon$ expressions

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*, then $\{\{\ e\ \}\}$ is of type $\{\tau\}$ *on the server*. Since no other typing rule involves client fragments, it is impossible to deconstruct them 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 a converter of type $\tau_s \rightsquigarrow \tau_c$ on the server, then $f\%e$ is of type τ_c on the client. Note that injections always require a converter, which we describe in greater details now.

Common rules

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

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

$$\textsc{App}\ \frac{\Gamma \triangleright_\ell e_1 : \tau_1 \to \tau_2 \qquad \Gamma \triangleright_\ell e_2 : \tau_1}{\Gamma \triangleright_\ell (e_1\ e_2) : \tau_2} \qquad \textsc{Y}\ \frac{}{\Gamma \triangleright_\ell \mathsf{Y} : ((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to \tau_1 \to \tau_2}$$

$$\textsc{QualVar}\ \frac{\Gamma \blacktriangleright_\ell p : (\mathtt{sig}\ S_1; \mathtt{val}_{\ell'}\ x_i : \tau; S_2\ \mathtt{end}) \qquad \ell' \succ \ell}{\Gamma \triangleright_\ell p.v : \tau[\mathrm{n_i} \mapsto_\ell \mathrm{p.n} \mid \mathrm{n_i} \in \mathrm{BV}_\ell(S_1)]}$$

Server rules

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

Client rules

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

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

Fig. 20. Eliom$_\varepsilon$ expression typing rules – $\Gamma \triangleright_\ell e : \tau$

3.3.2 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 $\tau_s \to \mathtt{serial}$, and a client-side decoding function of type $\mathtt{serial} \to \tau_c$. We assume the existence of two built-in converters:

- The `serial` converter of type $\mathtt{serial} \rightsquigarrow \mathtt{serial}$. Both sides are the identity.
- The `frag` converter of type $\forall \alpha_c.(\{\alpha_c\} \rightsquigarrow \alpha_c)$.

3.3.3 Type universes. It is important to note that there is no identity converter (that would be 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 example), are simply not transferable. For example, functions cannot be serialized in

$$\textsc{TypeVal}\ \frac{(\mathtt{type}_\ell\ (\alpha_{\ell_i})\mathtt{t}) \in \Gamma \qquad \forall i,\ \Gamma \vDash_{\ell_i} \tau_i}{\Gamma \vDash_\ell (\tau_i)\mathtt{t}} \qquad \textsc{ArrowVal}\ \frac{\Gamma \vDash_\ell \tau_1 \qquad \Gamma \vDash_\ell \tau_2}{\Gamma \vDash_\ell \tau_1 \to \tau_2} \qquad \textsc{VarVal}\ \frac{}{\Gamma \vDash_\ell \alpha_\ell} \qquad \textsc{FragVal}\ \frac{\Gamma \vDash_c \tau}{\Gamma \vDash_s \{\tau\}}$$

$$\textsc{QualifiedVal}\ \frac{\Gamma \blacktriangleright_\ell p : (\mathtt{sig}\ \mathcal{S}_1; \mathtt{type}_{\ell'}\ (\alpha_{\ell_i})\mathtt{t}; \mathcal{S}_2\ \mathtt{end}) \qquad \forall i,\ \Gamma \vDash_{\ell_i} \tau_i \qquad \ell' \succ \ell}{\Gamma \vDash_\ell (\tau_i)\mathrm{p.t}} \qquad \textsc{ConvVal}\ \frac{\Gamma \vDash_s \tau_1 \qquad \Gamma \vDash_c \tau_2}{\Gamma \vDash_s \tau_1 \rightsquigarrow \tau_2}$$

Fig. 21. Type validity rules – $\Gamma \vDash_\ell \tau$

$$\textsc{ReflEq}\ \frac{}{\Gamma \triangleright_\ell \tau \approx \tau} \qquad \textsc{TransEq}\ \frac{\Gamma \triangleright_\ell \tau_1 \approx \tau_2 \qquad \Gamma \triangleright_\ell \tau_2 \approx \tau_3}{\Gamma \triangleright_\ell \tau_1 \approx \tau_3} \qquad \textsc{CommEq}\ \frac{\Gamma \triangleright_\ell \tau_2 \approx \tau_1}{\Gamma \triangleright_\ell \tau_1 \approx \tau_2} \qquad \textsc{FunEq}\ \frac{\Gamma \triangleright_\ell \tau_1 \approx \tau_1' \qquad \Gamma \triangleright_\ell \tau_2 \approx \tau_2'}{\Gamma \triangleright_\ell \tau_1 \to \tau_2 \approx \tau_1' \to \tau_2'}$$

$$\textsc{DefTypeEq}\ \frac{(\mathtt{type}_{\ell'}\ (\alpha_{\ell_i})\mathtt{t} = \tau) \in \Gamma \qquad \ell' \succ \ell}{\Gamma \triangleright_\ell (\tau_i)\mathtt{t} \approx \tau\left[\alpha_i \mapsto_{\ell_i} \tau_i\right]_i} \qquad \textsc{AbsTypeEq}\ \frac{(\mathtt{type}_{\ell'}\ (\alpha_{\ell_i})\mathtt{t}) \in \Gamma \qquad \forall i,\ \Gamma \triangleright_{\ell_i} \tau_i \approx \tau_i' \qquad \ell' \succ \ell}{\Gamma \triangleright_\ell (\tau_i)\mathtt{t} \approx (\tau_i')\mathtt{t}}$$

$$\textsc{QualDefTypeEq}\ \frac{\Gamma \blacktriangleright_\ell p : (\mathtt{sig}\ \mathcal{S}_1; \mathtt{type}_{\ell'}\ (\alpha_{\ell_i})\mathtt{t} = \tau; \mathcal{S}_2\ \mathtt{end}) \qquad \forall i,\ \Gamma \triangleright_{\ell_i} \tau_i \approx \tau_i' \qquad \ell' \succ \ell}{\Gamma \triangleright_\ell (\tau_i)\mathrm{p.t} \approx \tau[n_i \mapsto_\ell p.n \mid n_i \in \mathrm{BV}_\ell(\mathcal{S}_1)]\left[\alpha_i \mapsto_{\ell_i} \tau_i\right]_i}$$

$$\textsc{QualAbsTypeEq}\ \frac{\Gamma \blacktriangleright_\ell p : (\mathtt{sig}\ \mathcal{S}_1; \mathtt{type}_{\ell'}\ (\alpha_{\ell_i})\mathtt{t}; \mathcal{S}_2\ \mathtt{end}) \qquad \forall i,\ \Gamma \triangleright_{\ell_i} \tau_i \approx \tau_i' \qquad \ell' \succ \ell}{\Gamma \triangleright_\ell (\tau_i)\mathrm{p.t} \approx (\tau_i')\mathrm{p.t}}$$

$$\textsc{FragmentEq}\ \frac{\Gamma \triangleright_c \tau \approx \tau'}{\Gamma \triangleright_s \{\tau\} \approx \{\tau'\}} \qquad \textsc{ConvEq}\ \frac{\Gamma \triangleright_s \tau_s \approx \tau_s' \qquad \Gamma \triangleright_c \tau_c \approx \tau_c'}{\Gamma \triangleright_s \tau_s \rightsquigarrow \tau_c \approx \tau_s' \rightsquigarrow \tau_c'}$$

Fig. 22. Type equivalence rules – $\Gamma \triangleright_\ell \tau \approx \tau'$

general. Another example is file handles: 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, as demonstrated in Section 2.3.2.

3.3.4 Mixed datatypes. The version of ML we consider supports an interesting combination of three features: abstract datatypes, parametrized datatypes and separate compilation at the module level. Eliom$_\varepsilon$, as an extension of ML, also supports these features. These three features have non-trivial interactions that need to be accounted for, in particular when introducing properties on types, such as locations.

Let us consider the module shown in Example 1. We declare a server datatype `t` with two parameters and we hide the definition in the signature. We now want to check that `(t1, t2)t` is a correct type expressions. However, without the type definition, we don't know if `t1` and `t2` are base, client or server types. In order to type check the type sub-expressions, we need more

```
module M : sig
  type%server ('a, 'b) t
end = struct
  type%server ('a, 'b) t = 'a fragment * 'b
end
```

(a) Incorrect abstract datatype

```
module M : sig
  type%server ('a[@client], 'b) t
end = struct
  type%server ('a[@client], 'b) t = 'a fragment * 'b
end
```

(b) Correct abstract datatype

Example 1. A module with an abstract datatype.
Figure 1a does not exposes information about acceptable sides for `'a` and `'b`. In Figure 1b, annotations specifying the side of type variables are exposed in the interface.

information about the definition of `t`. The solution, much like variance, is to annotate type variables in datatypes with extra information. This is done in the syntax for type declarations given in Figure 19. Each type parameters is annotated with a location. Type variables can only be used on the right location. This ensures proper separation of client and server type variables and their proper usage.

These annotations can be though as a simplistic kind system. One could also considered `'a` as a constrained type variable, in the style of ML^F [Botlan and Rémy 2003].

3.4 Modules

We now present the module language part of Eliom_ε as an extension of Leroy [1995] which model the bulk of the OCaml module language. The syntax, presented in Figure 23, is 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 difference between Eliom_ε and ML 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 m. We use ς for locations that can be m, b, s or c. Only modules on location m can have subfields on different locations. We also introduce mixed functors, noted $\texttt{functor}_m(X : \mathcal{M})\mathcal{M}$, which body can contain both client and server declarations. A program is a list of declarations including a client value declaration `return` which is the result of the program.

We first introduce the various feature of our module system along with some motivating examples. We then detail how those features are enforced by the typing rules.

3.4.1 Base location and specialization. In Section 2.1, 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 26, 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 function before being able to use it. Mixed modules only offer a limited version of polymorphism for locations: there is only one "location variable" at a time, and it's always called b. The specialization operation simply rewrites a module signature by substituting all instances of the location b or m 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

Module Expressions

$M ::= X_i \mid p.X$ (Variables)
$\mid (M : \mathcal{M})$ (Type constraint)
$\mid M_1(M_2)$ (Functor application)
$\mid \texttt{functor}(X_i : \mathcal{M})M$ (Functor)
$\mid \texttt{functor}_m(X_i : \mathcal{M})M$ (Mixed functor)
$\mid \texttt{struct}\ S\ \texttt{end}$ (Structure)

Structure body

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

Structure components

$D ::= \texttt{let}_\ell\ x_i = e$ (Values)
$\mid \texttt{type}_\ell\ \mathrm{t_i} = \tau$ (Types)
$\mid \texttt{module}_\varsigma\ X_i = M$ (Modules)

Programs

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

Module types

$\mathcal{M} ::= \texttt{sig}\ \mathcal{S}\ \texttt{end}$ (Signature)
$\mid \texttt{functor}(X_i : \mathcal{M}_1)\mathcal{M}_2$ (Functor)
$\mid \texttt{functor}_m(X_i : \mathcal{M}_1)\mathcal{M}_2$ (Mixed functor)

Signature body

$\mathcal{S} ::= \varepsilon \mid \mathcal{D}; \mathcal{S}$

Signature components

$\mathcal{D} ::= \texttt{val}_\ell\ x_i : \tau$ (Values)
$\mid \texttt{type}_\ell\ \mathrm{t_i} = \tau$ (Types)
$\mid \texttt{type}_\ell\ \mathrm{t_i}$ (Abstract types)
$\mid \texttt{module}_\varsigma\ X_i : \mathcal{M}$ (Modules)

Environments

$\Gamma ::= \mathcal{S}$

Fig. 23. Eliom$_m$'s grammar for modules

Figure 17. 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 m 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 the next section.

3.4.2 Mixed Functors. Mixed functors are functors that take as input a mixed module and return a mixed module. We note $\texttt{functor}_m(X_i : \mathcal{M})M$ the mixed functor that takes an argument X_i of type $\mathcal{M}$ 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. We saw in Section 2.4.4 an example of usage for mixed functors. Mixed functors have several restrictions compared to regular functors which we now detail using various examples.

Specialization. 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 Example 2, 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 Example 2b 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 is 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.

Injections. Injections inside client sections (as opposed to escaped values inside client fragments) are fairly static: the value might be dynamic, but the position of the injection and its use sites are statistically known and does not depend on the execution of the program. In particular, injections are independent of the control flow. We can just give a unique identifier to each injection, and use

```
module%mixed F (A : sig val b : int end)
= struct
  let%server x = A.b
  let%server y = [%client A.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 2. A mixed functor using base declaration polymorphically

that unique name for lookup on the client. This property comes from the fact that injected server identifiers cannot 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. Let us consider 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.

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

(a) An problematic mixed functor with an injection

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

(b) A pathological functor application

Example 3. Problematic example of injection inside a mixed functor

In order to avoid this situation, we add the constraints that injections inside the body of a mixed functors can only refer to outside of the functor. Escaped values, which are injections inside client fragments, are still allowed. The functor presented in Example 4a is not allowed while the one in Example 4b is allowed. Formally, this is guaranteed by the MixedFunctor rule, where each injection is typechecked in the outer typing environment.

```

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

```

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

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

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

Example 4. Mixed functor and injections

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 $\texttt{module}_m$ 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 3.6.

3.4.3 Type rules. We now review how these various language constructs are reflected in the rules of our type system. As before, the Eliom module system is built on the ML module system. We extend the typing, validity and subtyping judgments by a location annotation that specifies the location of the current scope. The program typing judgments don't have a location, since a program is always considered mixed. Most rules are fairly straightforward adaptations of the ML rules, annotated with locations.

The typing rules ModVar and QualModVar follow the usual rules of ML modules 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 $\varsigma' \succ \varsigma$ where ς is the current location and ς' 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 type of the identifier towards the current location ς. The specialization operation, which was described in Section 3.4.1, is noted $\lfloor M \rfloor_\varsigma$ and is defined in Figure 26.

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

3.4.4 Subtyping and equivalence of modules. Subtyping rules are given in Figure 25. For brevity, we note $\varsigma <: (\varsigma_1 \succ \varsigma_2)$ as a shorthand for $\varsigma <: \varsigma_1 \wedge \varsigma <: \varsigma_2 \wedge \varsigma_1 \succ \varsigma_2$, that is, both ς_1 and ς_2 are valid locations for components of a module on location ς and location ς_1 encompasses location ς_2. Note that the following holds:

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

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.

3.5 Separate typechecking

Our Eliom$_\varepsilon$ calculus inherit all the properties of the ML calculus regarding separate compilation. In particular, we show here that it trivially supports separate typechecking, where each module is typechecked independently and only requires knowledge of the module *types* (but not the implementations!) of its dependencies.

Theorem 1 (Separate Typechecking)**.** Given a list of module declarations that form a typed program, there exists an order such that each module can be typechecked with only knowledge of the type of the previous modules.

More formally, given a location ς and a list of n declarations D_i and a signature $\mathcal{S}$ such that

$$\blacktriangleright_\varsigma (D_1; \ldots; D_n) : \mathcal{S}$$

then there exists n definitions $\mathcal{D}_i$ and a permutation π such that

$$\forall \varsigma \forall i < n, \mathcal{D}_1; \ldots; \mathcal{D}_i \blacktriangleright_\varsigma D_{i+1} : \mathcal{D}_{i+1} \qquad \blacktriangleright_\varsigma \mathcal{D}_{\pi(1)}; \ldots; \mathcal{D}_{\pi(n)} <: \mathcal{S}$$

Proof. Proceed exactly as the ML proof for Theorem 5 in Appendix A.2.3. □

$$\textsc{ModVar}\ \frac{(\texttt{module}_{\varsigma'}\ X_i : \mathcal{M}) \in \Gamma \qquad \varsigma' \succ \varsigma}{\Gamma \blacktriangleright_\varsigma X_i : \lfloor \mathcal{M} \rfloor_\varsigma} \qquad \textsc{QualModVar}\ \frac{\Gamma \blacktriangleright_\varsigma p : (\texttt{sig}\ \mathcal{S}_1; \texttt{module}_{\varsigma'}\ X_i : \mathcal{M}; \mathcal{S}_2\ \texttt{end}) \qquad \varsigma' \succ \varsigma}{\Gamma \blacktriangleright_\varsigma p.X : \lfloor \mathcal{M}\left[n_i \mapsto_{\varsigma'} p.n \;\middle|\; n_i \in \mathrm{BV}_{\varsigma'}(\mathcal{S}_1)\right] \rfloor_\varsigma}$$

$$\textsc{Strength}\ \frac{\Gamma \blacktriangleright_\varsigma p : \mathcal{M}}{\Gamma \blacktriangleright_\varsigma p : \mathcal{M}/p} \qquad \frac{\Gamma \blacktriangleright_\varsigma M : \mathcal{M}' \qquad \Gamma \blacktriangleright_\varsigma \mathcal{M}' <: \mathcal{M}}{\Gamma \blacktriangleright_\varsigma M : \mathcal{M}}$$

$$\frac{\Gamma \blacktriangleright_\varsigma M_1 : \texttt{functor}(X_i : \mathcal{M})\mathcal{M}' \qquad \Gamma \blacktriangleright_\varsigma M_2 : \mathcal{M}}{\Gamma \blacktriangleright_\varsigma M_1(M_2) : \mathcal{M}'\left[X_i \mapsto_\varsigma M_2\right]}$$

$$\frac{\Gamma \vDash_\ell \mathcal{M} \qquad X_i \notin \mathrm{BV}_\ell(\Gamma) \qquad \Gamma; (\texttt{module}_\ell\ X_i : \mathcal{M}) \blacktriangleright_\ell M : \mathcal{M}'}{\Gamma \blacktriangleright_\ell \texttt{functor}(X_i : \mathcal{M})M : \texttt{functor}(X_i : \mathcal{M})\mathcal{M}'} \qquad \frac{\Gamma \vDash_\varsigma \mathcal{M} \qquad \Gamma \blacktriangleright_\varsigma M : \mathcal{M}}{\Gamma \blacktriangleright_\varsigma (M : \mathcal{M}) : \mathcal{M}}$$

$$\textsc{MixedFunctor}\ \frac{\Gamma \vDash_m \texttt{sig}\ \mathcal{S}\ \texttt{end} \qquad x_i \notin \mathrm{BV}_m(\Gamma) \qquad \Gamma; (\texttt{module}_m\ X_i : \texttt{sig}\ \mathcal{S}\ \texttt{end}) \blacktriangleright_m M : \mathcal{M}' \qquad \forall f_j \% X_j \in \mathrm{INJS}(M), \Gamma \triangleright_c f_j \% X_j : \tau_j}{\Gamma \blacktriangleright_m \texttt{functor}_m(X_i : \texttt{sig}\ \mathcal{S}\ \texttt{end})M : \texttt{functor}_m(X_i : \texttt{sig}\ \mathcal{S}\ \texttt{end})\mathcal{M}'}$$

$$\textsc{MixedApplication}\ \frac{\Gamma \blacktriangleright_\varsigma M_1 : \texttt{functor}_m(X_i : \mathcal{M})\mathcal{M}' \qquad \Gamma \blacktriangleright_m M_2 : \mathcal{M} \qquad m \succ \varsigma}{\Gamma \blacktriangleright_\varsigma M_1(M_2) : \mathcal{M}'\left[X_i \mapsto_\varsigma M_2\right]}$$

$$\frac{\Gamma \triangleright_\ell e : \tau \qquad x_i \notin \mathrm{BV}_\ell(\Gamma) \qquad \Gamma; (\texttt{val}_\ell\ x_i : \mathit{Close}(\tau, \Gamma)) \blacktriangleright_\varsigma S : \mathcal{S} \qquad \varsigma <: \ell}{\Gamma \blacktriangleright_\varsigma (\texttt{let}_\ell\ x_i = e; s) : (\texttt{val}_\ell\ x_i : \tau; \mathcal{S})}$$

$$\frac{\Gamma \vDash_\ell \tau \qquad \mathrm{t_i} \notin \mathrm{BV}_\ell(\Gamma) \qquad \Gamma; (\texttt{type}_\ell\ \mathrm{t_i} = \tau) \blacktriangleright_\varsigma S : \mathcal{S} \qquad \varsigma <: \ell}{\Gamma \blacktriangleright_\varsigma (\texttt{type}_\ell\ \mathrm{t_i} = \tau; s) : (\texttt{type}_\ell\ \mathrm{t_i} = \tau; \mathcal{S})}$$

$$\frac{\Gamma \blacktriangleright_\varsigma M : \mathcal{M} \qquad X_i \notin \mathrm{BV}_\varsigma(\Gamma) \qquad \Gamma; (\texttt{module}_\varsigma\ X_i : \mathcal{M}) \blacktriangleright_{\varsigma'} S : \mathcal{S} \qquad \varsigma' <: \varsigma \qquad \forall \varsigma'' \in \mathrm{locations}(\mathcal{M}).\ \varsigma'' \succ \varsigma}{\Gamma \blacktriangleright_{\varsigma'} (\texttt{module}_\varsigma\ X_i = M; s) : (\texttt{module}_\varsigma\ X_i : \mathcal{M}; \mathcal{S})}$$

$$\frac{\Gamma \blacktriangleright_\varsigma S : \mathcal{S}}{\Gamma \blacktriangleright_\varsigma \texttt{struct}\ S\ \texttt{end} : \texttt{sig}\ \mathcal{S}\ \texttt{end}} \qquad \frac{}{\Gamma \blacktriangleright_\varsigma \varepsilon : \varepsilon}$$

Fig. 24. Module typing rules – $\Gamma \blacktriangleright_\varsigma m : M$

3.6 Interpreted semantics

While Eliom, just like OCaml, is a compiled language, it is desirable to present a semantics that does not involve complex program transformation. The reason is two-fold: First, this simple semantics should be reasonably easy to explain to users. Indeed, this semantics is the one used to present Eliom in Section 2. However, we must also show that this semantics is correct, in that it does actually corresponds to our compilation scheme. This is done in Section 5.5. As presented in

$$\textsc{SubStruct}\quad \frac{\pi : [1;m] \to [1;n] \qquad \forall i \in [1;m],\ \Gamma; \mathcal{D}_1; \ldots; \mathcal{D}_n \blacktriangleright_\varsigma \mathcal{D}_{\pi(i)} <: \mathcal{D}'_i}{\Gamma \blacktriangleright_\varsigma (\mathtt{sig}\ \mathcal{D}_1; \ldots; \mathcal{D}_n\ \mathtt{end}) <: (\mathtt{sig}\ \mathcal{D}'_1; \ldots; \mathcal{D}'_m\ \mathtt{end})}$$

$$\frac{\Gamma \triangleright_{\ell_2} \tau_1 \approx \tau_2 \qquad \varsigma <: (\ell_1 \succ \ell_2)}{\Gamma \blacktriangleright_\varsigma (\mathtt{val}_{\ell_1}\ x_i : \tau_1) <: (\mathtt{val}_{\ell_2}\ x_i : \tau_2)} \qquad \frac{\Gamma \blacktriangleright_{\varsigma_2} \mathcal{M}_1 <: \mathcal{M}_2 \qquad \varsigma <: (\varsigma_1 \succ \varsigma_2)}{\Gamma \blacktriangleright_\varsigma (\mathtt{module}_{\varsigma_1}\ X_i : \mathcal{M}_1) <: (\mathtt{module}_{\varsigma_2}\ X_i = \mathcal{M}_2)}$$

$$\frac{\Gamma \blacktriangleright_\ell \mathcal{M}'_a <: \mathcal{M}_a \qquad \Gamma, (\mathtt{module}_\ell\ X : \mathcal{M}'_a) \blacktriangleright_\ell \mathcal{M}_r <: \mathcal{M}'_r}{\Gamma \blacktriangleright_\ell \mathtt{functor}(X : \mathcal{M}_a)\mathcal{M}_r <: \mathtt{functor}(X : \mathcal{M}'_a)\mathcal{M}'_r}$$

$$\frac{\Gamma \blacktriangleright_m \mathcal{M}'_a <: \mathcal{M}_a \qquad \Gamma, (\mathtt{module}_m\ X : \mathcal{M}'_a) \blacktriangleright_m \mathcal{M}_r <: \mathcal{M}'_r}{\Gamma \blacktriangleright_m \mathtt{functor}_m(X : \mathcal{M}_a)\mathcal{M}_r <: \mathtt{functor}_m(X : \mathcal{M}'_a)\mathcal{M}'_r}$$

$$\frac{\Gamma \triangleright_{\ell_2} \tau_1 \approx \tau_2 \qquad \varsigma <: (\ell_1 \succ \ell_2)}{\Gamma \blacktriangleright_\varsigma (\mathtt{type}_{\ell_1}\ \mathrm{t_i} = \tau_1) <: (\mathtt{type}_{\ell_2}\ \mathrm{t_i} = \tau_2)} \qquad \frac{\varsigma <: (\ell_1 \succ \ell_2)}{\Gamma \blacktriangleright_\varsigma (\mathtt{type}_{\ell_1}\ \mathrm{t_i}) <: (\mathtt{type}_{\ell_2}\ \mathrm{t_i})}$$

$$\frac{\Gamma \triangleright_{\ell_2} \mathrm{t_i} \approx \tau \qquad \varsigma <: (\ell_1 \succ \ell_2)}{\Gamma \blacktriangleright_\varsigma (\mathtt{type}_{\ell_1}\ \mathrm{t_i}) <: (\mathtt{type}_{\ell_2}\ \mathrm{t_i} = \tau)} \qquad \frac{\varsigma <: (\ell_1 \succ \ell_2)}{\Gamma \blacktriangleright_\varsigma (\mathtt{type}_{\ell_1}\ \mathrm{t_i} = \tau_1) <: (\mathtt{type}_{\ell_2}\ \mathrm{t_i})}$$

Fig. 25. Module subtyping rules – $\Gamma \blacktriangleright_\varsigma M <: M'$

$$\lfloor \mathcal{M} \rfloor_b = \mathcal{M} \qquad \lfloor \mathcal{M} \rfloor_m = \mathcal{M}$$

$$\lfloor \mathtt{sig}\ \mathcal{S}\ \mathtt{end} \rfloor_\iota = \mathtt{sig}\ \lfloor \mathcal{S} \rfloor_\iota\ \mathtt{end} \qquad \lfloor \mathtt{functor}_m(X_i : \mathcal{M})\mathcal{M}' \rfloor_\iota = \mathtt{functor}_m(X_i : \mathcal{M}) \lfloor \mathcal{M}' \rfloor_\iota$$

$$\lfloor \varepsilon \rfloor_\iota = \varepsilon \qquad \lfloor \mathtt{functor}(X_i : \mathcal{M})\mathcal{M}' \rfloor_\iota = \mathtt{functor}(X_i : \lfloor \mathcal{M} \rfloor_\iota) \lfloor \mathcal{M}' \rfloor_\iota$$

$$\lfloor \mathtt{val}_\ell\ x_i : \tau; \mathcal{S} \rfloor_\iota = \begin{cases} \mathtt{val}_\iota\ x_i : \tau; \lfloor \mathcal{S} \rfloor_\iota & \text{when } \ell \succ \iota \\ \lfloor \mathcal{S} \rfloor_\iota & \text{otherwise} \end{cases}$$

$$\lfloor \mathtt{type}_\ell\ \mathrm{t_i} = \tau; \mathcal{S} \rfloor_\iota = \begin{cases} \mathtt{type}_\iota\ \mathrm{t_i} = \tau; \lfloor \mathcal{S} \rfloor_\iota & \text{when } \ell \succ \iota \\ \lfloor \mathcal{S} \rfloor_\iota & \text{otherwise} \end{cases}$$

$$\lfloor \mathtt{type}_\ell\ \mathrm{t_i}; \mathcal{S} \rfloor_\iota = \begin{cases} \mathtt{type}_\iota\ \mathrm{t_i}; \lfloor \mathcal{S} \rfloor_\iota & \text{when } \ell \succ \iota \\ \lfloor \mathcal{S} \rfloor_\iota & \text{otherwise} \end{cases}$$

$$\lfloor \mathtt{module}_\varsigma\ X_i : \mathcal{M}; \mathcal{S} \rfloor_\iota = \begin{cases} \mathtt{module}_\iota\ X_i : \lfloor \mathcal{M} \rfloor_\iota; \lfloor \mathcal{S} \rfloor_\iota & \text{when } \varsigma \succ \iota \\ \lfloor \mathcal{S} \rfloor_\iota & \text{otherwise} \end{cases}$$

Where ι is either c or s.

Fig. 26. Module specialization operation – $\lfloor M \rfloor_\varsigma$

$$
\begin{aligned}
\varepsilon/p &= \varepsilon \\
(\texttt{sig}\ \mathcal{S}\ \texttt{end})/p &= \texttt{sig}\ \mathcal{S}/p\ \texttt{end} \\
(\texttt{module}_{\varsigma}\ X_i = \mathcal{M}; \mathcal{S})/p &= \texttt{module}_{\varsigma}\ X_i = \mathcal{M}/p; \mathcal{S}/p \\
(\texttt{type}_{\ell}\ \texttt{t}_\texttt{i} = \tau; \mathcal{S})/p &= \texttt{type}_{\ell}\ \texttt{t}_\texttt{i} = (\alpha^*)\texttt{p.t}; \mathcal{S}/p \\
(\texttt{type}_{\ell}\ \texttt{t}_\texttt{i}; \mathcal{S})/p &= \texttt{type}_{\ell}\ \texttt{t}_\texttt{i} = (\alpha^*)\texttt{p.t}; \mathcal{S}/p \\
(\texttt{val}_{\ell}\ x_i : \tau; \mathcal{S})/p &= \texttt{val}_{\ell}\ x_i : \tau; \mathcal{S}/p \\
\\
(\texttt{functor}(X_i : \mathcal{M})\mathcal{M}')/p &= \texttt{functor}(X_i : \mathcal{M})(\mathcal{M}'/p(X_i)) \\
(\texttt{functor}_m(X_i : \mathcal{M})\mathcal{M}')/p &= \texttt{functor}_m(X_i : \mathcal{M})(\mathcal{M}'/p(X_i))
\end{aligned}
$$

Fig. 27. Module strengthening operation – M/p

Section 2, Eliom execution proceeds in two steps: The server part of the program is executed first. This creates a client program, which is then executed.

Let us first introduce a few notations. Generated client programs are noted μ. Server expressions (resp. declarations) that do not contain injections are noted $\overline{e}$ (resp. $\overline{D}$). Values are the same as for ML: constants, closures, structures and functor closures. We consider a new class of identifiers called "references" and noted in bold, such as $\mathbf{r}$ or $\mathbf{R}$. We assume the existence of a name generator that can create arbitrary new fresh $\mathbf{r}$ identifiers at any point of the execution. References are used as global identifiers that ignore scoping rules. References can also be qualified as "reference paths", noted $\mathbf{X.r}$. This is used for mixed functors, in particular. We use γ to note the global environment where such references are stored.

We now introduce a new reduction relation, $\Rightarrow_{\varsigma}$, which is the reduction over Eliom constructs on side ς. The notation $\Rightarrow_{\varsigma}$ actually represents several reduction relations which are presented in Figures 29, 32 and 33. Four of these relations reduce the server part of the code and emit a client program. We note $e \overset{\rho}{\Longrightarrow}_{\iota} v, \mu, \theta$ the reduction of a server expression e inside a context ι in the environment ρ. It returns the value v, the client program μ and emits the trace θ. The context ι can be either base (b), server (s), server code inside client contexts (c/s) or server code inside mixed contexts (m). We also have a client reduction, noted $e \overset{\rho \mid \gamma \to \gamma'}{\Longrightarrow}_{c} v, \theta$ which reduces a client expression e inside an environment ρ, returns a value v and emits a trace θ. It also updates a global environment from γ to γ'.

Note that the first family of relation executes *only* the server part of a program and returns a client program, which is then executed by $\Rightarrow_c$. This is represented formally by the Program rule. In order to reduce an Eliom program P, we first reduce the server part using $\Rightarrow_m$. This returns no value and a client program μ which we execute. We now look into each specific feature in greater detail

3.6.1 Generated client programs. Let us first describe evaluation rules for generated client programs. Generated client programs are ML programs with some additional constructions which are described in Figure 28. The new evaluation rules are presented in Figure 29. The construction `bind env` $\mathbf{f}$ binds the current accessible environment to $\mathbf{f}$ in the global environment γ. This is implemented by the BindEnv rule. `bind` $\mathbf{r}$ `=` e `with` $\mathbf{f}$ computes e in the environment previously associated to $\mathbf{f}$. The result is then stored as $\mathbf{r}$ in γ. This construction is also usable for module expressions and is implemented by the Bind and Bind$_m$ rules. All these constructions also accept paths of references such as $\mathbf{R.f}$.

The new `bind` constructs are similar to the ones used in languages with continuations in the catch/throw style. Instead of storing both an environment and the future computation, we store only the environment. This will allow us to implement closures across locations, in particular the case where fragments are used inside a server closure.

The client reduction relation also inherits the ML rules (rule ClientCode). In such a case, the global environment is passed around following an order compatible with traces. For example, the LetIn rule for let expression would be modified like so:

$$\frac{e' \xrightarrow{\rho \mid \gamma \to \gamma'}_c v', \theta \qquad e \xrightarrow{\rho + \{x \mapsto v'\} \mid \gamma' \to \gamma''}_c v, \theta'}{(\texttt{let}\ x = e'\ \texttt{in}\ e) \xrightarrow{\rho \mid \gamma \to \gamma''}_c v, \theta \mathbin{@} \theta'}$$

Here, e' is evaluated first (since θ is present first in the resulting traces), hence it uses the initial environment γ and returns the environment γ', which is then passed along.

From now on, we use $\mathbf{f}$ to denote the reference associated to fragments closures and $\mathbf{r}$ to denote the reference associated to a specific value of a fragment.

$$\begin{aligned}
\mathbf{p} &::= (\mathbf{X}.)^*\mathbf{x} && \text{(Reference path)}\\
D_c &::= D_{\mathrm{ML}} \\
&\mid \texttt{bind env}\ \mathbf{p} && \text{(Env binding)}\\
&\mid \texttt{bind}\ \mathbf{p} = e\ \texttt{with}\ \mathbf{p}' \mid \texttt{bind}\ \mathbf{p} = M\ \texttt{with}\ \mathbf{p}' && \text{(Global binding)}
\end{aligned}$$

Fig. 28. Grammar of client programs

$$\textsc{Bind}\quad \frac{e \xrightarrow{\gamma(\mathbf{p_f}) \mid \gamma \to \gamma'}_c v, \theta \qquad S \xrightarrow{\rho \mid (\gamma' + \{\mathbf{p} \mapsto v\}) \to \gamma''}_c V, \theta'}{(\texttt{bind}\ \mathbf{p} = e\ \texttt{with}\ \mathbf{p_f}; S) \xrightarrow{\rho \mid \gamma \to \gamma''}_c V, \theta \mathbin{@} \theta'}$$

$$\textsc{BindEnv}\quad \frac{S \xrightarrow{\rho \mid (\gamma + \{\mathbf{p_f} \mapsto \rho\}) \to \gamma'}_c V, \theta}{(\texttt{bind env}\ \mathbf{p_f}; S) \xrightarrow{\rho \mid \gamma \to \gamma'}_c V, \theta}$$

$$\textsc{Bind}_m\quad \frac{M \xrightarrow{\gamma(\mathbf{p_F}) \mid \gamma \to \gamma'}_c V, \theta \qquad S \xrightarrow{\rho \mid (\gamma' + \{\mathbf{p} \mapsto V\}) \to \gamma''}_c V', \theta'}{(\texttt{bind}\ \mathbf{p} = M\ \texttt{with}\ \mathbf{p_F}; S) \xrightarrow{\rho \mid \gamma \to \gamma''}_c V', \theta \mathbin{@} \theta'}$$

ClientCode
Inherit the rules from ML

Fig. 29. Semantics for client generated programs – $e \xrightarrow{\rho \mid \gamma \to \gamma'}_c v, \theta$

3.6.2 Base, Client and Server declarations. We now consider the case of base, client and server declarations. The rules are presented in Figure 32. Let us first describe the execution of complete Eliom$_\varepsilon$ programs (rule Program). A program P reduces to a client value v if and only if we can first create a *server* reduction of P that produces no value, emits a client program μ and a trace θ_s. We can then create a reduction of μ that reduces in v with a trace θ_c. The trace of the program is the concatenation of the traces. We see that the execution of Eliom$_\varepsilon$ program is split in two as described earlier. Let us now look in more details at various construction of the Eliom$_\varepsilon$ language.

Base. The base reduction relation corresponds exactly to the ML reduction relation, and always returns empty programs (rule BaseCode). When reducing a base declaration in a mixed context, we both reduce the declaration using $\Rightarrow_b$, but also add the declaration to the emitted client program (rule BaseDecl). As we can see, base declarations are executed twice: once on the server and once on the client.

Client contexts and injections. The goal of the client reduction relation $\Rightarrow_{c/s}$ is *not* to reduce client programs. It only reduces server code contained by injections inside client code. It returns a client expression without injections, a client program and a trace. Since we don't want to execute client code, it does not inherit the reduction rules for ML. Given an injection $f\%e$, the rule Injection reduces the server side expression $(f^s\ e)$ to a value v. We then transform the server value v into a client value using the $\downarrow$ operator presented Figure 31. We then returns the client expression $(f^c\ \downarrow v)$ without executing it. This expression will be executed on the client side, to deserialize the value. The value injection operator, noted $\downarrow$ represents the serialization of values from the server to the client and is the identity over constants in $Const_b$ and references, and fail on any other values. According to the definition of converters, if f is a converter $\tau_s \rightsquigarrow \tau_c$, then f^s is the server side function of type $\tau_s \rightarrow \texttt{serial}$ and v should be of type `serial`. Since `serial` is defined on b, the injection of values should be the identity.

The rule ClientContext defines the evaluation of server expression up to client contexts. Client contexts are noted $E[e_1, \ldots, e_n]$ and are defined in Figure 30. A client context can have any number of holes which must all contain injections. The rest of the context can contain arbitrary ML syntax that are not injections. Evaluation under a multi-holed context proceed from left to right. The resulting programs and traces are constructed by concatenation of each program and trace.

In order to evaluate client declarations, the rule ClientDecl uses $\Rightarrow_{c/s}$ to evaluate the server code present in the declaration D_c which returns a declaration without injections $\overline{D_c}$ and a client program μ. We then return the client program composed by the concatenation of the two. We demonstrate this in Example 5. The Eliom$_\varepsilon$ program is presented on the left side. It first declares the integer a on the server then inject it on the client and returns the result. The emitted code, shown in the middle, contains an explicit call to the $\texttt{int}^c$ deserializer while the rest of the client code is unchanged. The returned value is shown on the right.

$$\begin{array}{l}\texttt{let}_s\ a = 3\\ \texttt{let}_c\ \texttt{return} = \texttt{int}\%x + 1\end{array} \quad \Longrightarrow_m \quad \texttt{let return} = (\texttt{int}^c\ 3) + 1 \quad \Longrightarrow_c \quad 4, \langle\rangle$$

Example 5. Execution of a client declaration

$$\begin{array}{rcl} E_e & ::= & [f\%e] \mid e \mid (E_e\ E_e) \mid \lambda x.E_e \mid \texttt{let}\ x = E_e\ \texttt{in}\ E_e \\ E_M & ::= & M \mid (E_M : \mathcal{M}) \mid E_M(E_M) \\ & & \mid \texttt{functor}(X_i : \mathcal{M})E_M \mid \texttt{struct}\ (E_D)^*\ \texttt{end} \\ E_D & ::= & D \mid \texttt{let}_c\ x_i = E_e \mid \texttt{module}_c\ X_i = E_M \end{array}$$

Fig. 30. Execution contexts for injections – $E[\cdot]$

$$\begin{array}{ll} \downarrow c = c & \text{when } c \in Const_b \\ \downarrow \mathbf{p} = \mathbf{p} & \\ \downarrow v = \bot & \text{otherwise} \end{array}$$

Fig. 31. Injections of values – $\downarrow v$

Server code and fragments. The server reduction relation reduces server code and emits the appropriate client program associated to client fragments. Since client program are mostly ML programs, it inherits the ML reduction rules (rule ServerCode) where client programs are concatenated in the same order as traces. Client fragments are handled by the rule Fragment. Let us consider a fragment $\{\{\ e\ \}\}$, this evaluation proceeds in two steps: first, we evaluate all the injections inside the client expression e using the relation $\Rightarrow_{c/s}$ described in the previous section. We thus obtain an expression without injection $\overline{e}$ and a client program μ.

The second step is to register $\overline{e}$ to be evaluated in the client program. One could propose to simply consider client fragments as values. This is however quite problematic, as it could lead to

duplicated side effects. Consider the program presented on the left side of Example 6. If we were simply to simply pass fragments along, the `print` statement would be evaluated twice. Instead, we create a fresh identifier $\mathbf{r}$ that will be globally bound to $\overline{e}$ in the client program, as shown in rule FRAGMENT. This way, the client expression contained inside the fragment will be executed once, in a timely manner. The execution rule for fragment is demonstrated in Example 6. As before, the ELIOM$_\varepsilon$ program is presented on the left, the emitted client program in shown in the middle and the returned value is on the right. Note that both $\texttt{frag}^s$ and $\texttt{frag}^c$ are the identity function.

$$\begin{array}{l}\texttt{let}_s\ x = \{\{\ (\texttt{print}\ 3)\ \}\}\\ \texttt{let}_c\ \texttt{return} =\\ \quad \texttt{frag\%}x + \texttt{frag\%}x\end{array} \quad \Longrightarrow_m \quad \begin{array}{l}\texttt{bind env}\ \mathbf{f}\\ \texttt{bind}\ \mathbf{r} = (\texttt{print}\ 3)\ \texttt{with}\ \mathbf{f}\\ \texttt{let return} =\\ \quad (\texttt{frag}^c\ \mathbf{r}) + (\texttt{frag}^c\ \mathbf{r})\end{array} \quad \Longrightarrow_c \quad 6, \langle 3 \rangle$$

Example 6. Execution of a fragment containing side-effects

Closures and fragments. In the client program above, we also use a reference $\mathbf{f}$ and the `bind env` construct. To see why this is necessary, we now consider a case where fragments are used inside closures. This is presented in Example 7. The ELIOM$_\varepsilon$ program, presented on the left, computes $1 + 3 + 2$ on the client (although in a fairly obfuscated way). We first define the client variable a as 1. We then define a server closure f containing a client fragment capturing a. We then define a new variable also named a and call $(f\ 3)$, inject the results and returns. When evaluating the definitions of f, since it contains syntactically a client fragment, we will emit the client instruction `bind env` $\mathbf{f}$, where $\mathbf{f}$ is a fresh identifier. This will capture the local environment, which is $\{a \mapsto 1\}$ at this point of the client program. When we execute $(f\ 3)$, we will finally reduce the client fragment and emit the $(\texttt{bind}\ \mathbf{r} = (\texttt{int}^c\ 3) + a\ \texttt{with}\ \mathbf{f})$ instruction. On the client, this will be executed in the $\mathbf{f}$ environment, hence a is 1 and the result is 4. Once this is executed, we move back to the regular environment, where a is 2, and proceed with the execution.

Thanks to this construction, the capturing behavior of closures is preserved across location boundaries. The `bind env` construct is generated by the SERVERDECL rule. FRAGS(D_s) returns the fragments syntactically present in D_s. For each fragment, the local environment is bound to the associated reference.

$$\begin{array}{l}\texttt{let}_c\ a = 1\\ \texttt{let}_s\ f\ x = \{\{\ \texttt{int\%}x + a\ \}\}\\ \texttt{let}_c\ a = 2\\ \texttt{let}_s\ y = (f\ 3)\\ \texttt{let}_c\ \texttt{return} = \texttt{frag\%}y + a\end{array} \quad \Longrightarrow_m \quad \begin{array}{l}\texttt{let}\ a = 1\\ \texttt{bind env}\ \mathbf{f}\\ \texttt{let}\ a = 2\\ \texttt{bind}\ \mathbf{r} = (\texttt{int}^c\ 3) + a\ \texttt{with}\ \mathbf{f}\\ \texttt{let return} = (\texttt{frag}^c\ \mathbf{r}) + a\end{array} \quad \Longrightarrow_c \quad 6, \langle\rangle$$

Example 7. Execution of a fragment inside a closure

Fragment annotations. In the previous examples, we presented the server reduction rules where, for each syntactic fragment, a fresh reference $\mathbf{f}$ is generated and bound to the environment. In the rest of this thesis, we will simply assume that all fragments syntactically present in the program are annotated with a unique reference. Such annotation is purely syntactic and can be done by walking the syntax tree of the program. Annotated fragments are noted $\{\{\ \ldots\ \}\}_{\mathbf{f}}$.

Mixed structures syntactically present in the program are also annotated in a similar manner with a unique module reference. Annoted mixed structures are noted $\texttt{struct}\ \ldots\ \texttt{end}_{\mathbf{F}}$.

3.6.3 Mixed modules. Let us now describe the reduction relation for mixed modules. The mixed reduction relation is presented in Figure 33 and, just like the server relation, has for goal to evaluate all the server code and emit a client program to be later evaluated by the client relation. Mixed modules can be composed of either mixed functors, functor applications or structures. The mixed relation contains various rules that are similar to the ML reduction rules for modules. The notable novel aspect of mixed functor is that they both have a client part and a server part. This is different from client fragments, which only have a client part that can be manipulated on the server via an identifier. The server part of mixed modules also need to indicate its client part. In order to do this, each mixed structure will contains an additional field called `Dyn` which contains a module identifier. The identifier points to a globally bound module on the client which is the result of the client-side evaluation.

Let us first demonstrate these features in Example 8. In this example, we declare a mixed module X containing a fragment x and an integer y. We then declare another mixed module Y containing a submodule. The structure of the emitted client code mimics closely the structure of the server code. In particular, the `bind` operation is nested inside the mixed module X that is emitted on the client. The exact same names are reused on the client. We also register each structure in the global environment using the annotated identifier of the structure. Here, we use the `bind` construct as a shorthand for `bind with` that doesn't change the environment. The shape of the program is kept intact thanks to the MixedModVar, MixedQualModVar and MixedStruct rules. The first

Server code inside client contexts

Injection

$$\frac{(f^s\ e) \overset{\rho}{\Longrightarrow}_s v, \mu, \theta}{f\%e \overset{\rho}{\Longrightarrow}_{c/s} (f^c\ \downarrow v), \mu, \theta}$$

ClientContext

$$\frac{\forall i,\ e_i \overset{\rho}{\Longrightarrow}_{c/s} v_i, \mu_i, \theta_i}{E[e_1, \ldots, e_n] \overset{\rho}{\Longrightarrow}_{c/s} E[v_1, \ldots, v_n], \mu_1; \ldots; \mu_n, @_i\ \theta_i}$$

Server code

Fragment

$$\frac{e \overset{\rho}{\Longrightarrow}_{c/s} \overline{e}, \mu, \theta \qquad \mathbf{r} \text{ fresh}}{\{\!\{\ e\ \}\!\}_{\mathrm{f}} \overset{\rho}{\Longrightarrow}_s \mathbf{r}, (\mu; \texttt{bind r = } \overline{e} \texttt{ with f}), \theta}$$

ServerCode
Inherit the rules from ML

Base code

BaseCode

$$\overset{\rho}{\Longrightarrow} \equiv \overset{\rho}{\Longrightarrow}_b$$

Declarations

BaseDecl

$$\frac{D_b \overset{\rho}{\Longrightarrow}_b V, \varepsilon, \theta \qquad S \overset{\rho+\mathrm{V}}{\Longrightarrow}_m V', \mu', \theta'}{D_b; S \overset{\rho}{\Longrightarrow}_m V + V', (D_b; \mu'), \theta \mathbin{@} \theta'}$$

ClientDecl

$$\frac{D_c \overset{\rho}{\Longrightarrow}_{c/s} \overline{D_c}, \mu, \theta \qquad S \overset{\rho}{\Longrightarrow}_m V, \mu', \theta'}{D_c; S \overset{\rho}{\Longrightarrow}_m V, (\mu; \overline{D_c}; \mu'), \theta}$$

ServerDecl

$$\frac{\mathrm{FRAGS}(D_s) = \overline{\{\!\{\ e_i\ \}\!\}_{\mathrm{f}_i}} \qquad D_s \overset{\rho}{\Longrightarrow}_s V, \mu, \theta \qquad S \overset{\rho+\mathrm{V}}{\Longrightarrow}_m V', \mu', \theta'}{D_s; S \overset{\rho}{\Longrightarrow}_m V + V', (\overline{\texttt{bind env f}_\mathbf{i}}; \mu; \mu'), \theta \mathbin{@} \theta'}$$

Program

$$\frac{P \overset{\rho}{\Longrightarrow}_m (), \mu, \theta_s \qquad \mu \overset{\rho \mid \varepsilon \rightarrow \gamma}{\Longrightarrow}_c v, \theta_c}{P \overset{\rho}{\Longrightarrow} v, \theta_s \mathbin{@} \theta_c}$$

Fig. 32. Semantics for base, client and server sections – $e \overset{\rho}{\Longrightarrow}_\varsigma v, \mu, \theta$

two are similar to the non mixed version, but the last one deserves some explanation. First, it prefixes all the fragment references inside the body of the structure. This is for consistency with functors, as we will see later. It then adds the `Dyn` field to the returned structure, as discussed before. Finally, it emits a `bind` on the client and returns the module reference. Each structure is thus bound appropriately, even when nested.

Module identifiers are not used in the present program, but they are used in the case of mixed functors, as we will see now.

$$
\begin{array}{l}
\texttt{module}_m\ X = \texttt{struct} \\
\quad \texttt{let}_s\ x = \{\{\ 1\ \}\} \\
\quad \texttt{let}_c\ y = 2 + \texttt{frag\%}x \\
\texttt{end}_{\mathbf{X}} \\
\texttt{module}_m\ Y = \texttt{struct} \\
\quad \texttt{module}_m\ A = X \\
\texttt{end}_{\mathbf{Y}} \\
\texttt{let}_c\ \texttt{return} = Y.A.y
\end{array}
\Longrightarrow_m
\begin{array}{l}
\texttt{bind}\ \mathbf{X} = \texttt{struct} \\
\quad \texttt{bind env}\ \mathbf{X.f} \\
\quad \texttt{bind}\ \mathbf{r} = 1\ \texttt{with}\ \mathbf{X.f} \\
\quad \texttt{let}\ y = 2 + (\texttt{frag}^c\ \mathbf{r}) \\
\texttt{end} \\
\texttt{module}\ X = \mathbf{X} \\
\texttt{bind}\ \mathbf{Y} = \texttt{struct} \\
\quad \texttt{module}\ A = X \\
\texttt{end} \\
\texttt{module}\ Y = \mathbf{Y} \\
\texttt{let return} = Y.A.y
\end{array}
\Longrightarrow_c \quad 3, \langle\rangle
$$

Example 8. Execution of mixed modules

$$
\begin{array}{l}
\texttt{let}_s\ x = 1 \\
\texttt{module}_m\ F(X : \mathcal{M}) = \texttt{struct} \\
\quad \texttt{let}_c\ b = X.a + \texttt{int\%}x \\
\texttt{end}_{\mathbf{Y}} \\
\texttt{module}_m\ Y = \texttt{struct} \\
\quad \texttt{let}_c\ a = 2 \\
\texttt{end}_{\mathbf{Y}} \\
\texttt{module}_c\ Z = F(Y) \\
\texttt{let}_c\ \texttt{return} = Z.b
\end{array}
\Longrightarrow_m
\begin{array}{l}
\texttt{module}\ F(X : \mathcal{M}) = \texttt{struct} \\
\quad \texttt{let}\ b = X.a + (\texttt{int}^c\ 1) \\
\texttt{end}_{\mathbf{Y}} \\
\texttt{bind}\ \mathbf{Y} = \texttt{struct} \\
\quad \texttt{let}\ a = 2 \\
\texttt{end} \\
\texttt{module}\ Y = \mathbf{Y} \\
\texttt{module}\ Z = F(Y) \\
\texttt{let return} = Z.b
\end{array}
\Longrightarrow_c \quad 3, \langle\rangle
$$

Example 9. Execution of mixed functors with injections

Mixed functors, injections and client side application. Before exposing the complex interaction of mixed functors and fragment, let us illustrate various details about mixed functors in Example 9. The server code proceed in the following way: we first define a server variable x followed by a mixed functor F containing an injection. We then define a mixed module Y and executes *on the client* the functor application $F(Y)$.

First, let us recall that injections inside mixed functors can only refer to elements outside the functor. This means that injections inside functors can be reduced as soon as we consider a functor. In particular, we do not wait for functor application. This can be seen in the ModClosure rule which returns a functor closure on the server side and emit the client part of the functor on the client side. We then take the client part of the body of the functor (noted $M|_c$) and applies the $\Rightarrow_{c/s}$ reduction relation, which executes injections inside client code. In this example, it results in the injection `int%`x being resolved immediately in the client-side version of the functor.

Mixed functor application can be done in client and server contexts. When it is done in a client context, we simply call the client-side definition and omits the server-side execution completely.

$$
\begin{array}{l}
\texttt{module}_m\ F(X:\mathcal{M}) = \texttt{struct} \\
\quad \texttt{let}_s\ x = \{\{\ X.a + \texttt{int\%}X.b\ \}\}_{\mathbf{f_x}} \\
\texttt{end}_\mathbf{F} \\
\texttt{module}_m\ Y = \texttt{struct} \\
\quad \texttt{let}_c\ a = 4 \\
\quad \texttt{let}_s\ b = 2 \\
\texttt{end}_\mathbf{Y} \\
\\
\texttt{module}_m\ Z = F(Y) \\
\\
\texttt{let}_c\ \texttt{return} = \texttt{frag\%}Z.x
\end{array}
\Longrightarrow_m
\begin{array}{l}
\texttt{bind env}\ \mathbf{F} \\
\texttt{module}\ F(X:\mathcal{M}) = \texttt{struct end} \\
\texttt{bind}\ \mathbf{Y} = \texttt{struct} \\
\quad \texttt{let}\ a = 4 \\
\texttt{end} \\
\texttt{module}\ Y = \mathbf{Y} \\
\\
\texttt{bind}\ \mathbf{R_Z} = \texttt{struct} \\
\quad \texttt{module}\ X = \mathbf{Y} \\
\quad \texttt{bind env}\ \mathbf{R_Z.f_x} \\
\quad \texttt{bind}\ \mathbf{r_x} = Y.a + (\texttt{int}^c\ 2) \\
\quad\ \ \texttt{with}\ \mathbf{R_Z.f_x} \\
\texttt{end with}\ \mathbf{F} \\
\texttt{module}\ Z = F(Y) \\
\\
\texttt{let return} = (\texttt{frag}^c\ \mathbf{r_x})
\end{array}
\Longrightarrow_c 6, \langle\rangle
$$

Example 10. Execution of mixed functors with fragments

Hence we can simply emit the client-code $F(Y)$. Execution is done through the usual rules for client sections. This is always valid since each mixed declaration emits a client declaration with the same name and the same shape.

Mixed functors and fragments. The difficulty of the reduction of mixed functor containing fragments is that the server-side application of a mixed functor should result in both server and client effects. This makes the reduction rules for mixed functor application quite delicate. We illustrate this with Example 10. In this example, we define a functor F contains only the server declaration x. The argument of the functor simply contains two integers, one on the server and one on the client. In the fragment bound to x, we add the two integers (using an escaped value). The interesting aspect here is that the body of the client fragment depends on both the client and the server side of the argument, even if there is no actual client side for the functor F. The rest of the program is composed of a simple mixed module Y and the mixed functor application $F(Y)$.

The first step of the execution is to define the client side part of F and Y, as demonstrated in the previous example. In this case, since F only contains a server side declaration, the client part of the functor returns an empty structure. We then have to execute $F(Y)$. This is done with the StructBeta rule. When reducing a mixed functor application, we first generate a fresh identifier ($\mathbf{R_Z}$ here) and prefix all the fragment closure identifiers. We then evaluate the body of the functor on the server, which gives us both the server module value and the generated client code. In this case, we simply obtain the binding of $\mathbf{r_x}$. Note that this reference is not prefixed by $\mathbf{R_Z}$ since it is freshly generated at runtime. If the functor was applied again, we would simply generated a new one. In order for functor arguments to be properly available on the client, we need to introduce additional bindings. For this purpose, we lookup the `Dyn` field for each module argument and insert the additional binding. In this case, `module` $X = \mathbf{Y}$. This gives us a complete client structure which we can bind to $\mathbf{R_Z}$.

We see here that the body of functors allows to emit client code in a dynamic but controlled way. Generated module references used on the client are remembered on the server using the `Dyn` field while closure identifiers ensure that the proper environment is used. One problematic aspect

Mixed module expressions

$$\textsc{MixedStruct}\ \frac{S[\mathbf{f_i} \mapsto \mathbf{X}.\mathbf{f_i}]_i \overset{\rho}{\Longrightarrow}_m V, \mu, \theta \qquad V' = V + \{\mathtt{Dyn} \mapsto \mathbf{X}\}}{\mathtt{struct}\ S\ \mathtt{end}_{\mathbf{X}} \overset{\rho}{\Longrightarrow}_m V', \mathbf{X}, \mathtt{bind}\ \mathbf{X} = \mathtt{struct}\ \mu\ \mathtt{end}, \theta} \qquad \textsc{MixedModVar}\ \frac{\rho(X) = V}{X \overset{\rho}{\Longrightarrow}_m V, X, \varepsilon, \langle\rangle}$$

$$\textsc{App}\ \frac{M \overset{\rho}{\Longrightarrow}_m V, M_c, \mu, \theta \qquad M' \overset{\rho}{\Longrightarrow}_m V', M'_c, \mu', \theta' \qquad V(V') \overset{\rho}{\Longrightarrow}_m V'', \mu'', \theta''}{M(M') \overset{\rho}{\Longrightarrow}_m V'', M_c(M'_c), \mu; \mu'; \mu'', \theta \mathbin{@} \theta' \mathbin{@} \theta''}$$

$$\textsc{StructBeta}\ \frac{\begin{array}{l}\mathbf{R}\ \text{fresh} \qquad V_f = \mathtt{functor}_m(\rho')(X_i : \mathcal{M}_i)_i \mathtt{struct}\ S\ \mathtt{end}_{\mathbf{F}} \\ V_i(\mathtt{Dyn}) = \mathbf{R_i} \qquad S[\mathbf{f_i} \mapsto \mathbf{R}.\mathbf{f_i}]_i \overset{\rho' + \{X_i \mapsto V_i\}_i}{\Longrightarrow}_m V, \mu, \theta\end{array}}{V_f(V_1) \ldots (V_n) \overset{\rho}{\Longrightarrow}_m V + \{\mathtt{Dyn} \mapsto \mathbf{R}\}, \begin{pmatrix}\mathtt{bind}\ \mathbf{R} = \mathtt{struct} \\ \quad (\mathtt{module}\ X_i = \mathbf{R_i};)_i \\ \quad \mu \\ \mathtt{end\ with}\ \mathbf{F}\end{pmatrix}, \theta}$$

$$\textsc{NotStructBeta}\ \frac{V = \mathtt{functor}_m(\rho')(X_i : \mathcal{M}_i)_i M \qquad M \overset{\rho' + \{X_i \mapsto V_i\}_i}{\Longrightarrow}_m V_r, \mu, \theta}{V(V_1) \ldots (V_n) \overset{\rho}{\Longrightarrow}_m V_r, \mu, \theta,} \qquad \textsc{MixedQualModVar}\ \frac{p \overset{\rho}{\Longrightarrow}_m V, \mu, \theta}{p.X \overset{\rho}{\Longrightarrow}_m V(X), p.X, \mu, \theta} \qquad \textsc{Empty}\ \frac{}{\varepsilon \overset{\rho}{\Longrightarrow}_m \{\}, \varepsilon, \langle\rangle}$$

$$\textsc{ModClosure}\ \frac{M|_c \overset{\rho}{\Longrightarrow}_{c/s} \overline{M}, \mu, \theta}{\mathtt{functor}_m(X : \mathcal{M})M \overset{\rho}{\Longrightarrow}_m \mathtt{functor}_m(\rho)(X : \mathcal{M})M, \mathtt{functor}(X : \mathcal{M}|_c)\overline{M}, \mu, \theta}$$

Mixed declarations

$$\textsc{MixedModDecl}\ \frac{\mathrm{MixedStructIds}(M) = \overline{\mathbf{F_i}} \qquad M \overset{\rho}{\Longrightarrow}_m V, M^c, \mu, \theta \qquad S \overset{\rho + \{X \mapsto V\}}{\Longrightarrow}_m V', \mu', \theta'}{\mathtt{module}_m\ X = M;\ S \overset{\rho}{\Longrightarrow}_m \{X \mapsto V\} + V', (\overline{\mathtt{bind\ env}\ \mathbf{F_i}}; \mu; \mathtt{module}\ X = M^c; \mu'), \theta \mathbin{@} \theta'}$$

Fig. 33. Semantics for mixed modules – $M \overset{\rho}{\Longrightarrow}_{\varsigma} V, \mu, \theta$

of this method is that it leads to two executions of the client side. A partial work-around is to keep track of loaded client module to avoid duplicating side-effects.

4 INTEGRATION WITH OCAML

We now formalize more precisely the interaction between Eliom$_\varepsilon$, its location system, and the vanilla ML language. In particular, Theorem 2 shows that ML modules can be completely embedded in Eliom$_\varepsilon$ programs without changes.

Let us note $\mathcal{M}_{[\ell \mapsto \ell']}$ the substitution on locations in an Eliom module type $\mathcal{M}$. Given an ML module M, we note $M_{[\mathrm{ML} \mapsto \ell]}$ the Eliom module where all the module components have been annotated with location ℓ. Given en Eliom module M, we note $M_{[\mapsto \mathrm{ML}]}$ the ML module where all the location have been erased. We extend these notations to module types and environments.

4.1 Properties of specialization

Let us first clarify the behavior of specialization with regard to mono-located Eliom$_\varepsilon$ module types.

Proposition 1. Given an Eliom module type $\mathcal{M}$ and a location $\ell \in \{b, c, s\}$, if $\Gamma \vDash_\ell \mathcal{M}$, then $\lfloor \mathcal{M} \rfloor_\ell = \mathcal{M}$.

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

Proposition 2. Given an Eliom module type $\mathcal{M}$ and a location $\ell \in \{c, s\}$, if $\Gamma \vDash_b \mathcal{M}$, then $\lfloor \mathcal{M} \rfloor_\ell = \mathcal{M}_{[b \mapsto \ell]}$.

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

4.2 Interaction between ML and Eliom$_\varepsilon$

We can now relate ML code to equivalent Eliom$_\varepsilon$ code that has been annotated with a single location.

Proposition 3. Given ML type τ, expression e, module M and module type $\mathcal{M}$ and locations ℓ, ℓ':

$$\begin{aligned}
\Gamma \vDash_{\mathrm{ML}} \tau &\implies \Gamma_{[\mathrm{ML} \mapsto \ell']} \vDash_\ell \tau && \text{Where } \ell' \succ \ell \\
\Gamma \triangleright_{\mathrm{ML}} e : \tau &\implies \Gamma_{[\mathrm{ML} \mapsto \ell']} \triangleright_\ell e : \tau && \text{Where } \ell' \succ \ell \\
\Gamma \vDash_{\mathrm{ML}} \mathcal{M} &\implies \Gamma_{[\mathrm{ML} \mapsto \ell']} \vDash_\ell \mathcal{M}_{[\mathrm{ML} \mapsto \ell]} && \text{Where } \ell' \succ \ell \\
\Gamma \blacktriangleright_{\mathrm{ML}} M : \mathcal{M} &\implies \Gamma_{[\mathrm{ML} \mapsto \ell']} \blacktriangleright_\ell M_{[\mathrm{ML} \mapsto \ell]} : \mathcal{M}_{[\mathrm{ML} \mapsto \ell]} && \text{Where } \ell' \succ \ell
\end{aligned}$$

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

- By Proposition 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 $\ell' \succ \ell$ are respected by hypothesis. □

Proposition 4. Given ML type τ, expression e, module m and module type $\mathcal{M}$:

$$\begin{aligned}
\Gamma \vDash_b \tau &\implies \Gamma_{[\mapsto \mathrm{ML}]} \vDash_{\mathrm{ML}} \tau & \qquad \Gamma \vDash_b \mathcal{M} &\implies \Gamma_{[\mapsto \mathrm{ML}]} \vDash_{\mathrm{ML}} \mathcal{M}_{[\mapsto \mathrm{ML}]} \\
\Gamma \triangleright_b e : \tau &\implies \Gamma_{[\mapsto \mathrm{ML}]} \triangleright_{\mathrm{ML}} e : \tau & \qquad \Gamma \blacktriangleright_b M : \mathcal{M} &\implies \Gamma_{[\mapsto \mathrm{ML}]} \blacktriangleright_{\mathrm{ML}} M_{[\mapsto \mathrm{ML}]} : \mathcal{M}_{[\mapsto \mathrm{ML}]}
\end{aligned}$$

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 coincides with ML. We can then proceed by induction over the proof trees. □

Proposition 5. Given an ML module M (resp. expression e), an execution environment ρ, a location ℓ, a value V (resp. v) and a trace θ:

$$\begin{aligned}
M \overset{\rho}{\Longrightarrow} V, \theta &\iff M_{[\mathrm{ML} \mapsto \ell]} \overset{\rho_{[\mathrm{ML} \mapsto \ell]}}{\Longrightarrow}_\ell V_{[\mathrm{ML} \mapsto \ell]}, \varepsilon, \theta \\
e \overset{\rho}{\Longrightarrow} v, \theta &\iff e_{[\mathrm{ML} \mapsto \ell]} \overset{\rho_{[\mathrm{ML} \mapsto \ell]}}{\Longrightarrow}_\ell v_{[\mathrm{ML} \mapsto \ell]}, \varepsilon, \theta
\end{aligned}$$

Furthermore, given an ML program P, an execution environment ρ, a value v and a trace θ:

$$P \overset{\rho}{\Longrightarrow} v, \theta \iff P_{[\mathrm{ML}\mapsto\iota]} \overset{\rho_{[\mathrm{ML}\mapsto\iota]}}{\Longrightarrow} v_{[\mathrm{ML}\mapsto\iota]}, \theta \qquad \iota \in \{c, s\}$$

$$P \overset{\rho}{\Longrightarrow} v, \theta \iff P_{[\mathrm{ML}\mapsto b]} \overset{\rho_{[\mathrm{ML}\mapsto\mathrm{b}]}}{\Longrightarrow} v_{[\mathrm{ML}\mapsto b]}, \theta \mathbin{@} \theta$$

Proof. Let us first note that the ML reduction relation is included in the base, the server and the client-only relations. Additionally, the considered programs, modules or expressions can not contain fragments, injections or binds. The additional rules in the server and client-only relations are only used for these additional syntactic constructs. For the first three statements, we can then proceed by induction. For the last statement, we remark that base code is completely copied to the client during server execution. Using rule Program, we execute the program twice, which returns the same value but duplicates the trace. □

Theorem 2 (Base/ML correspondance)**.** Eliom modules, expressions and types on base location b correspond exactly to the ML language.

Proof. By Propositions 3, 4 and 5. □

Thanks to Theorem 2, we can completely identify the language ML and the part of Eliom on base location. This is of course by design: the base location allows us to reason about the host language, OCaml, inside the new language Eliom. It also provides the guarantee that anything written in the base location does not contain any communication between client and server. In the rest of the thesis, we omit location substitutions of the form $_{[\mathrm{ML}\mapsto b]}$ and $_{[b\mapsto\mathrm{ML}]}$.

Proposition 3 also has practical consequences: Given a file previously typechecked by an ML 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. This give us the possibility, in the implementation, to completely reuse compiled objects from the OCaml typechecker and load them on an arbitrary location. In particular, it guarantees that we can reuse pure OCaml libraries safely and freely.

5 COMPILATION OF ELIOM PROGRAMS

In Section 3, we gave a tour of the Eliom$_\varepsilon$ language from a formal perspective, providing a type system and an interpreted semantics. Eliom, however, is not an interpreted language. The interpreted semantics is here both as a formal tool and to make the semantics of the language more approachable to users, as demonstrated in Section 2. In the implementation, Eliom programs are compiled to 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). The resulting programs are efficient and avoid unnecessary back-and-forth communications between the client and the server.

Description of the complete compilation toolchain, including emission of JavaScript code, is out of scope of this article (see Vouillon and Balat [2014]). Instead, we describe the compilation process in term of emission of client and server programs in an ML-like language equipped with additional primitives. Hence, we present the typing and execution of compiled programs, in Section 5.1 and the compilation process, in Section 5.2. We then show that our compilation process preserves both the typing and the semantics in Section 5.5.

5.1 Target languages $\mathrm{ML_s}$ and $\mathrm{ML_c}$

We introduce the two target languages $\mathrm{ML_c}$ and $\mathrm{ML_s}$ as extensions of ML. The additions in these two new languages are highlighted in Figure 34. Typing is provided in Section 5.1.5. The semantics

is provided in Section 5.1.6. As before, we use globally bound identifiers, which we call "references" and note in bold: **r**. References can also be paths, such as **X.r**. In some contexts, we accept a special form of reference path, noted `Dyn.`**x** which we explain in Section 5.1.4. In practice, these references are implemented with uniquely generated names and associative tables. Contrary to the interpreted semantics, references are also used to transfer values from the server to the client and can appear in expressions. A reference used inside an expression is always of type `serial`.

ML_s grammar		**ML_c grammar**	
$\mathbf{p}$ ::= `Dyn.`$\mathbf{f}$ | $(\mathbf{X}.)^*\mathbf{f}$	(Reference path)	$\mathbf{p}$ ::= `Dyn.`$\mathbf{f}$ | $(\mathbf{X}.)^*\mathbf{f}$	(Reference path)
τ ::= ... | `frag`	(Fragment type)	e ::= ... | $\mathbf{x}$	(Reference)
v ::= ... | $\mathbf{r}$	(Reference)	M ::= ... | $\mathbf{X}$	(Module reference)
e ::= ...		D ::= ...	
| `fragment` $\mathbf{p}$ e^*	(Fragment call)	| `bind` $\mathbf{p} = e$	(Fragment closure)
M ::= ...		| $\texttt{bind}_m$ $\mathbf{p} = M$	(Functor fragment)
| p.`Dyn`	(Dynamic field)	| `exec ()`	(Fragment execution)
| $\texttt{fragment}_m$ $\mathbf{p}$ (M^*)	(Fragment)		
D ::= ...			
| `injection` $\mathbf{x}$ e	(Injection)		
| `end ()`	(End token)		

Fig. 34. Grammar for ML_s and ML_c as extensions of ML_ε

5.1.1 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 $\tau_s \rightarrow$ `serial` and f^c is of type `serial` $\rightarrow \tau_c$. We will generally assume that if the converter f is available in the environment, then f^c and f^s are available in the client and server environment respectively.

5.1.2 Injections. For injections, 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 over, a mapping from references to injected values is sent to the client. $\mathbf{v}$ is then used client-side to access the value.

An example is given in Example 11. In this example, two integers are sent from the server to the client and add them on the client. We suppose the existence of a base abstract type `int`, a converter `int` of type (`int` $\rightsquigarrow$ `int`) and the associated encoding and decoding functions. The server program, in Example 11a, creates two injections, $\mathbf{v_1}$ and $\mathbf{v_2}$ and does not expose any bindings nor return any values. These injections hold the serialized integers 4 and 2. The client program, in Example 11b, uses these two injections, deserialize their values, adds them, and returns the result. Note that `injection` *is not a network operation.* It simply stores a mapping between references (*i.e.*, names) and serialized values. The mapping generated at the end of the server execution is shown in Example 11c. After the execution of the server code, this mapping is sent to the client, and used to execute the client code.

5.1.3 Fragments. The primitive related to fragments also relies on shared references between the server program and the client program. However, these references allow to uniquely identify

```
injection v1 (int^s 4);
injection v2 (int^s 2);
```
(a) Server program

```
let return =
  (int^c v1) + (int^c v2);
```
(b) Client program

$\mathbf{v_1} \mapsto 4$
$\mathbf{v_2} \mapsto 2$
(c) Mapping of injections

Example 11. Client-server programs calling and using injections

functions that are defined on the client but are called on the server. To implement this, we use the following primitives:

- In ML_C structures, `bind` $\mathbf{p} = e$ declares a new client function bound to the reference $\mathbf{p}$. The function e takes an arbitrary amount of argument of type `serial` and returns any type.
- In ML_S expressions, `fragment` $\mathbf{p}\ e_1 \ldots e_n$ is a delayed function application which registers that, on the client, the function associated to $\mathbf{p}$ will be applied to the arguments e_i. 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.

Here again, none of these primitives are network communication primitives. While the API is similar to Remote Procedure Calls, the execution is very different: `fragment` only accumulates the function call in a list, to be executed later. When the server execution is over, the list of calls is sent to the client, and used during the client execution. OCaml, and consequently Eliom, are impure languages: the order of execution of statement is important. In order to control the order of execution, we introduce two additional statements: `end ()`, on the server, introduces an `end` marker in the list of calls. `exec ()`, on the client, executes all the calls until the next `end` token.

Example 12 presents a pair of programs which emit the client trace $\langle 2; 3; 3\rangle$, but in such a way that, while the client does the printing, the values and the execution order are completely determined by the server. The server code (Example 12a) calls $\mathbf{f}$ with 2 as argument, injects the result and then calls $\mathbf{f}$ with 3 as argument. The client code, in Example 12b, declares a fragment closure $\mathbf{f}$, which simply adds one to its arguments, and exec both fragments. In-between both executions, it prints the content of the injection $\mathbf{v}$. During the execution of the server, the list of calls (Example 12c) and the mapping of injections (Example 12d) are built. First, when `fragment` $\mathbf{f}$ `(int`^{s} 2) is executed, a fresh reference $\mathbf{r_1}$ is generated, the call to the fragment is added to the list and $\mathbf{r_1}$ is returned. The injection adds the association $\mathbf{v_1} \mapsto \mathbf{r_1}$ to the mapping of injections. The call to `end ()` then adds the token `end` to the list of fragments. The second fragment proceeds similarly to the first, with a fresh identifiers $\mathbf{r_2}$. Once server execution is over, the newly generated list of fragments and mapping of injections are sent to the client. During the client execution, the execution of the list is controlled by the `exec` calls. First, ($\mathbf{f}$ 2) emits $\langle 2\rangle$ and is evaluated to 3, and the mapping $\mathbf{r_1} \mapsto 3$ is added to a global environment. Then $\mathbf{v_1}$ is resolved to $\mathbf{r_1}$ and printed (which shows $\langle 3\rangle$). Finally ($\mathbf{f}$ 3) emits $\langle 3\rangle$ and is evaluated to 4.

The important thing to note here is that both the injection mapping and the list of fragments are completely dynamic. We could add complicated control flow to the server program that would drive the client execution according to some dynamic values. The only static elements are the names $\mathbf{f}$ and $\mathbf{v_1}$, the behavior of $\mathbf{f}$ and the number of call to `exec ()`. We cannot, however, make the server-side control flow depend on client-side values, since we obey a strict phase separation between server and client execution.

Finally, remark that we do not need the `bind env` construct introduced in Section 3.6.1. Instead, we directly capture the environment using closures that are extracted in advance. We will see how this extraction works in more details while studying the compilation scheme, in Section 5.2.

```
let x1 = fragment f (int^s 2);
injection v1 (frag^s x1);
end ();
let x2 = fragment f (int^s 3);
end ();
```

(a) Server program

```
bind f = λx.((print (int^c x)) + 1);
exec ();
let a = (print (frag^c v1));
exec ();
let return = a
```

(b) Client program

$\{\mathbf{r_1} \mapsto (\mathbf{f}\ 2)\}$; end;
$\{\mathbf{r_2} \mapsto (\mathbf{f}\ 3)\}$; end;
(c) List of fragments

$\mathbf{v_1} \mapsto \mathbf{r_1}$
(d) Mapping of injections

Example 12. Client-server program defining and calling fragments

5.1.4 Modules. We introduce three new module-related construction that are quite similar to fragment primitives:

- $\texttt{bind}_m\ \mathbf{p} = M$ is equivalent to `bind` for modules. It is a client instruction that associates the module or functor M to the reference $\mathbf{p}$.
- $\texttt{fragment}_m\ \mathbf{p}\ (\mathbf{R_1}) \ldots (\mathbf{R_n})$ is analogous to `fragment` $\mathbf{p}\ e$ for modules. It is a delayed functor application that is used on the server to register that the functor associated to $\mathbf{p}$ will have to be applied to the modules associated to $\mathbf{R_i}$. 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$_\varepsilon$ mixed structure that have both a server and a client part.

The first argument of `fragment`, $\texttt{fragment}_m$, `bind` and $\texttt{bind}_m$ can also be a reference path Dyn.$\mathbf{f}$, where `Dyn` is the locally bound `Dyn` field inside a module. This allows us to isolate some bound references inside a fresh module reference. This is useful for functors, as we will now demonstrate in Example 13.

In this example, we again add integers[2] on the client while controlling the values and the control flow on the server. We want to define server modules that contain server values but also trigger some evaluation on the client, in a similar way to fragments. The first step is to define a module X on the server and to bind a corresponding module $\mathbf{X}$ on the client. Similarly to the interpreted semantics presented in Section 3.6, we add a `Dyn` field to the server module that points to the client module. Plain structures such as X are fairly straightforward, as we only need to declare each part statically and add the needed reference. $\texttt{bind}_m$ allows to declare modules globally.

We then declare the functor F on the server and bind the functor $\mathbf{F}$ on the client. The server-side functor contains a call to a fragment defined in the client-side functor. The difficulty here is that we should take care of differentiating between fragment closures produced by different functor applications. For this purpose, we use a similar technique than the one presented in Section 3.6.3, which is to prefix the fragment closure identifier $\mathbf{f}$ with the reference of the client-side module. This reference is available on the server side as the `Dyn` field and is generated by a call to $\texttt{fragment}_m$. When F is applied to X on the server, we generate a fresh reference $\mathbf{R}$ and add $\{\mathbf{R_1} \mapsto \mathbf{F}\ \mathbf{X_0}\}$ to the execution queue. When `exec ()` is called, We introduce the additional binding $\{\texttt{Dyn} \mapsto \mathbf{R_1}\}$ in the environment and apply $\mathbf{F}$ to $\mathbf{X_0}$, which will register the $\mathbf{R_1}.\mathbf{f}$ fragment closure. Since it is the result of this specific functor application, the closure $\mathbf{R_1}.\mathbf{f}$ will always add 4 to its argument. The rest of the execution proceed as shown in the previous section: we call a new fragment, which triggers the client-side addition 2 + 4 and use an injection to pass the results around.

5.1.5 Type system rules. The ML_s and ML_c typing rules are presented in Figures 35 and 36 as a small extension over the ML typing rules presented in Appendix A.2. Note that the typing rules for the new primitives are weakly typed and are certainly not sound with respect to serialization and

[2]But better! or at very least, more obfuscated.

$$\textsc{Fragment}\ \frac{\forall i,\ \Gamma \triangleright_{\mathrm{ML_s}} e_i : \texttt{serial}}{\Gamma \triangleright_{\mathrm{ML_s}} \texttt{fragment}\ \mathbf{p}\ e_1\ \ldots\ e_n : \texttt{frag}} \qquad \textsc{Injection}\ \frac{\Gamma \triangleright_{\mathrm{ML_s}} e : \texttt{serial}}{\Gamma \blacktriangleright_{\mathrm{ML_s}} \texttt{injection}\ \mathbf{v}\ e : \varepsilon} \qquad \textsc{End}\ \frac{}{\Gamma \blacktriangleright_{\mathrm{ML_s}} \texttt{end}\ () : \varepsilon}$$

$$\textsc{Fragment}_m\ \frac{\forall i,\ \Gamma \triangleright_{\mathrm{ML_s}} p_i : \mathcal{M}_\mathrm{i}}{\Gamma \blacktriangleright_{\mathrm{ML_s}} \texttt{module Dyn} = \texttt{fragment}_m\ \mathbf{p}\ p_1.\texttt{Dyn}\ \ldots\ p_n.\texttt{Dyn} : \varepsilon}$$

Fig. 35. Typing rules for $\mathrm{ML_s}$

$$\textsc{Bind}\ \frac{\Gamma \triangleright_{\mathrm{ML_c}} e : \tau}{\Gamma \blacktriangleright_{\mathrm{ML_c}} \texttt{bind}\ \mathbf{p} = e : \varepsilon} \qquad \textsc{Bind}_m\ \frac{\Gamma \blacktriangleright_{\mathrm{ML_c}} M : \mathcal{M}}{\Gamma \blacktriangleright_{\mathrm{ML_c}} \texttt{bind}_m\ \mathbf{p} = M : \varepsilon} \qquad \textsc{Reference}\ \frac{}{\Gamma \triangleright_{\mathrm{ML_c}} \mathbf{x} : \texttt{serial}} \qquad \textsc{Exec}\ \frac{}{\Gamma \blacktriangleright_{\mathrm{ML_c}} \texttt{exec}\ () : \varepsilon}$$

Fig. 36. Typing rules for $\mathrm{ML_s}$

deserialization. Given arbitrary $\mathrm{ML_s}$ and $\mathrm{ML_c}$ programs, there is no guarantee that (de)serialization will not fail at runtime. This is on purpose. Indeed, all these guarantees are provided by Eliom itself. $\mathrm{ML_s}$ and $\mathrm{ML_c}$ are target languages that are very liberal by design, so that all patterns permitted by Eliom are expressible with them. Furthermore, from an implementation perspective. $\mathrm{ML_s}$ and $\mathrm{ML_c}$ are simply OCaml libraries and do not rely on further compiler support. Note that `Dyn` fields are not reflected in signatures. The fragment $\textsc{Fragment}_m$ rule does not enforce that the `Dyn` field is present in all the arguments. This is enforced by construction during compilation.

5.1.6 Semantics rules. We define two reduction relations as extensions of the ML reduction rules (see Appendix A.3). The $\Rightarrow_{\mathrm{ML_s}}$ reduction for $\mathrm{ML_s}$ server programs is presented in Figure 37. The $\Rightarrow_{\mathrm{ML_c}}$ reduction for $\mathrm{ML_c}$ client programs is presented in Figure 38. Let us consider a server structure S_s and a client structure S_c. A paired execution of the two structures is presented below:

$$S_s \overset{\rho_s}{\Longrightarrow}_{\mathrm{ML_s}} V_s, \xi, \zeta, \theta_s \qquad\qquad S_c, \xi \overset{\rho_c \mid \gamma \cup \zeta \to \gamma'}{\Longrightarrow}_{\mathrm{ML_c}} V_c, \xi', \theta_c$$

Let us now detail these executions rules. As with the ML reduction, ρ_s and ρ_c are the local environments of values while θ_s and θ_c are the traces for server and client executions respectively. V_s and

```
module X = struct
  module Dyn = X0;
  let a = 2
end;
module F(Y : Ms) = struct
  module Dyn = fragment_m F (Y.Dyn);
  let b = fragment Dyn.f (int^s Y.a);
end;
module Z = F(X);
end ();
injection v1 (frag^s Z.b);
```

(a) Server program

```
bind_m X0 = struct
  let c = 4
end;

bind_m F(Y : Mc) = struct
  bind Dyn.f =
    λa.((int^c a) + Y.c);
end;

exec ()
let return = (frag^c v1);
```

(b) Client program

$\{\mathbf{R_1} \mapsto \mathbf{F}\ \mathbf{X_0}\}$;
$\{\mathbf{r_2} \mapsto \mathbf{R_1}.\mathbf{f}\ 2\}$;
end

(c) List of fragments

$\mathbf{v_1} \mapsto \mathbf{r_2}$

(d) Mapping of injections

Example 13. Client-server program using module fragments

V_c are the returned values. Similarly to the interpreted semantics for Eliom$_\varepsilon$, the client reduction uses global environments noted γ.

As we saw in the previous examples, server executions emits two sets of information during execution: a queue of fragments and a map of injections. Mapping of injection is a traditional (global) environment where bindings are noted $\{\mathbf{v}\mapsto\dots\}$. The queue of fragments is noted ξ and contains end tokens end and fragment calls $\{\mathbf{r}\mapsto\mathbf{f}\ v_1\dots v_n\}$. Concatenation of fragment queues is noted $+\!\!+$. We now see the various rules in more details.

Injections. Injection bindings are collected on the server through the Injection rule. When creating a new injection binding, we inject the server-side value using the injection of value operator, noted $\downarrow v$ and presented in Figure 31. This models the serialization of values before transmission from server to client by ensuring that only base values and references are injected. Other kinds of values should be handled using converters explicitly.

The injection environment ζ forms a valid client-side global environment. When executing the client-side program, we simply assume that ζ is included in the initial global environment γ.

Fragments and functors. On the server, fragments and functors calls are added to the queue through the Fragment and Fragment$_m$ rules. In both rules, the reference of the associated closure or functor is provided, along with a list of arguments. A fresh reference symbolizing the fragment is generated and the call is added to the queue ξ. Note that in the case of regular fragments, the arguments are expressions which can themselves contains fragment calls. The module rule Fragment$_m$ is similar, the main difference being that it only accept module references as arguments of the call.

Fragment closures and functors are bound on the client through the Bind and Bind$_m$ rules, which simply binds a reference to a value or a module value in the global environment γ. Since `bind` accepts references of the form `Dyn.`**f**, it must first resolves `Dyn` to the actual reference. This is done through the Dyn rule.

Segmented execution. In ML$_\text{s}$ and ML$_\text{c}$ programs, the execution of fragments is segmented through the use of the `exec ()`/`end ()` instructions. On the server, `end` instructions are handled through the End rule, which simply adds an `end` token to the execution queue ξ. On the client, we use the Exec rule, associated to the Frag and Frag$_m$ rules. When `exec` is called, the Exec rule triggers the execution of the segment of the queue until the next `end` token. Each token $\{\mathbf{r}\mapsto\mathbf{f}\ v_1\dots v_n\}$ is executed with the Frag rule as the function call $(\mathbf{f}\ v_1\dots v_n)$. The result of this function call is bound to **r** in the global environment γ. Similarly, functor calls are executed using the Frag$_m$ rule. Note that for functors we also introduce the `Dyn` field in the local environment, which allows local `bind` definitions. Once all the tokens have been executed in the considered fragment queue, we resume the usual execution.

5.2 Slicing

In Section 5.1, we presented the two target languages ML$_\text{s}$ and ML$_\text{c}$. We now present the compilation process transforming *one* Eliom$_\varepsilon$ program into two distinct ML$_\text{s}$ and ML$_\text{c}$ programs. Before giving a more formal description in Section 5.3, we present the compilation process through three examples of increasing complexity.

Injections and fragments. Example 14 presents an Eliom$_\varepsilon$ program containing only simple declarations involving fragments and injections without modules. The Eliom$_\varepsilon$ program is presented on the left, while the compiled ML$_\text{s}$ and ML$_\text{c}$ programs are presented on the right. In this example, a first fragment is created. It only contains an integer and is bound to a. A second fragment that

Injection

$$\frac{e \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} v, \xi, \zeta, \theta \qquad S \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} V, \xi', \zeta', \theta'}{\texttt{injection}\ \mathbf{x}\ e; S \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} V, \xi +\!\!+ \xi', (\zeta \cup \{\mathbf{x} \mapsto \downarrow v\} \cup \zeta'), \theta\ @\ \theta'}$$

Fragment

$$\frac{\mathbf{p} \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} \mathbf{p}' \qquad \forall i,\ e_i \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} v_i, \xi_i, \zeta_i, \theta_i \qquad \mathbf{r}\ \text{fresh}}{\texttt{fragment}\ \mathbf{p}\ e_1\ \ldots\ e_n \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} \mathbf{r}, (\xi_1 +\!\!+ \ldots \xi_n +\!\!+ \{\mathbf{r} \mapsto \mathbf{p}'\ \downarrow v_1 \ldots \downarrow v_n\}), \cup_i \zeta_i, @_i\ \theta_i}$$

Fragment$_m$

$$\frac{\mathbf{p} \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} \mathbf{p}' \qquad \forall i,\ p_i.\mathsf{Dyn} \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} \mathbf{R}_i, \varepsilon, \varepsilon, \langle\rangle \qquad \mathbf{R}\ \text{fresh}}{\texttt{fragment}_m\ \mathbf{p}\ p_1.\mathsf{Dyn}\ \ldots\ p_n.\mathsf{Dyn} \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} \mathbf{R}, \{\mathbf{R} \mapsto \mathbf{p}'\ \mathbf{R}_1 \ldots \mathbf{R}_n\}, \{\}, \langle\rangle}$$

End

$$\frac{S \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} V, \xi, \zeta, \theta}{\mathsf{end}\ (); S \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} V, \mathsf{end} +\!\!+ \xi, \zeta, \theta}$$

Dyn

$$\frac{\rho(\mathsf{Dyn}) = \mathbf{R}}{\mathsf{Dyn.f} \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} \mathbf{R}.\mathbf{r}}$$

ServerCode
Inherit the rules from ML

Fig. 37. Semantics rules for $\mathsf{ML_s}$– $S \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} V, \xi, \zeta, \theta$

Bind

$$\frac{\mathbf{p} \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} \mathbf{p}' \qquad e, \xi \overset{\rho \mid \gamma \to \gamma'}{\Longrightarrow}_{\mathrm{ML_c}} v, \xi', \theta \qquad S, \xi' \overset{\rho \mid \gamma' + \{\mathbf{p}' \mapsto \mathrm{v}\} \to \gamma''}{\Longrightarrow}_{\mathrm{ML_c}} V, \xi'', \theta'}{(\texttt{bind}\ \mathbf{p} = e; S), \xi \overset{\rho \mid \gamma \to \gamma''}{\Longrightarrow}_{\mathrm{ML_c}} V, \xi'', \theta\ @\ \theta'}$$

Bind$_m$

$$\frac{\mathbf{p} \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} \mathbf{p}' \qquad M, \xi \overset{\rho \mid \gamma \to \gamma'}{\Longrightarrow}_{\mathrm{ML_c}} V_{\mathbf{p}}, \xi', \theta \qquad S, \xi' \overset{\rho \mid \gamma' + \{\mathbf{p}' \mapsto \mathrm{V_p}\} \to \gamma''}{\Longrightarrow}_{\mathrm{ML_c}} V, \xi'', \theta'}{(\texttt{bind}_m\ \mathbf{p} = M; S), \xi \overset{\rho \mid \gamma \to \gamma''}{\Longrightarrow}_{\mathrm{ML_c}} V, \xi'', \theta\ @\ \theta'}$$

Reference

$$\frac{\gamma(\mathbf{r}) = v}{\mathbf{r}, \xi \overset{\rho \mid \gamma \to \gamma}{\Longrightarrow}_{\mathrm{ML_c}} v, \xi, \langle\rangle}$$

Exec

$$\frac{\xi \overset{\rho \mid \gamma \to \gamma'}{\Longrightarrow}_{\mathrm{ML_c}} \{\}, \theta \qquad S, \xi' \overset{\rho \mid \gamma' \to \gamma''}{\Longrightarrow}_{\mathrm{ML_c}} V, \xi'', \theta'}{(\mathsf{exec}\ (); S), \xi +\!\!+ \mathsf{end} +\!\!+ \xi' \overset{\rho \mid \gamma \to \gamma''}{\Longrightarrow}_{\mathrm{ML_c}} V, \xi'', \theta\ @\ \theta'}$$

Frag

$$\frac{(\mathbf{f}\ v_1 \ldots v_n) \overset{\rho \mid \gamma \to \gamma'}{\Longrightarrow}_{\mathrm{ML_c}} v, \theta \qquad \xi \overset{\rho \mid \gamma' + \{\mathbf{r} \mapsto \mathrm{v}\} \to \gamma''}{\Longrightarrow}_{\mathrm{ML_c}} \{\}, \theta'}{\{\mathbf{r} \mapsto \mathbf{f}\ v_1 \ldots v_n\} +\!\!+ \xi \overset{\rho \mid \gamma \to \gamma'}{\Longrightarrow}_{\mathrm{ML_c}} \{\}, \theta\ @\ \theta'}$$

Dyn

$$\frac{\rho(\mathsf{Dyn}) = \mathbf{R}}{\mathsf{Dyn.f} \overset{\rho}{\Longrightarrow}_{\mathrm{ML_c}} \mathbf{R}.\mathbf{r}}$$

Frag$_m$

$$\frac{\mathbf{F}(\mathbf{R}_1) \ldots (\mathbf{R}_n) \overset{\rho \cup \{\mathsf{Dyn} \mapsto \mathbf{R}\} \mid \gamma \to \gamma'}{\Longrightarrow}_{\mathrm{ML_c}} V, \theta \qquad \xi \overset{\rho \mid \gamma' + \{\mathbf{R} \mapsto \mathrm{V}\} \to \gamma''}{\Longrightarrow}_{\mathrm{ML_c}} \{\}, \theta'}{\{\mathbf{R} \mapsto \mathbf{F}\ \mathbf{R}_1 \ldots \mathbf{R}_n\} +\!\!+ \xi \overset{\rho \mid \gamma \to \gamma'}{\Longrightarrow}_{\mathrm{ML_c}} \{\}, \theta\ @\ \theta'}$$

ClientCode
Inherit the ML rules

Fig. 38. Semantics rules for $\mathsf{ML_c}$– $S, \xi \overset{\rho \mid \gamma \to \gamma'}{\Longrightarrow}_{\mathrm{ML_c}} V, \xi', \theta$

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 emit a `bind` declaration on the client. 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. Similarly to the interpreted semantics, we use the client part of the converter on the client. In this case, {{ 1 }} is turned into $\lambda().1$ and {{ frag%a + 1 }} is turned into $\lambda v.((\texttt{frag}^c\ v) + 1)$. On the server, each fragment is replaced by a call to the primitive `fragment`. The arguments of the call are the identifier of the closure and all the injections that are contained in the fragment. The `fragment` primitive, which was presented in Section 5.1.3, registers that the closure declared on the client 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. We place an `exec`/`end` pair at each server section. This is enough to ensure that client code inside server fragment and client code in regular client declaration is executed in the expected order.

Note that injections, which occur outside of fragments, and escaped values, which occur inside fragments, are compiled in a 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$_\varepsilon$	ML$_S$	ML$_C$
let$_s$ a = {{ 1 }};	let a = fragment $\mathbf{f_0}$ ();	bind $\mathbf{f_0}$ = $\lambda().1$;
	end ();	exec ();
let$_s$ b = {{ frag%a + 1 }};	let b = fragment $\mathbf{f_1}$ (frag^s a);	bind $\mathbf{f_1}$ = $\lambda v.((\texttt{frag}^c\ v) + 1)$;
	end ();	exec ();
let$_c$ return = frag%b + 2;	injection $\mathbf{x}$ (frag^s b);	let return = (frag^c $\mathbf{x}$) + 2;

Example 14. Compilation of expressions

Sections and modules. We now present an example with client and server modules in Example 15. The lines of the various programs have been laid out in a way that should highlight their correspondence.

We declare a server module X containing a client fragment, a client functor F containing an injection, a client functor application Z and finally the client return value, with another injection. The compilation of the server module X proceeds in the following way: on the server, we emit a module X similar to the original declaration but where fragments have been replaced by a call to the `fragment` primitive. On the client, we only emit the call to `bind`, without any of the server-side code structure. Compilation for the rest of the code proceeds in a similar manner.

This compilation scheme is correct thanks to the following insight: In client and server modules or functors, the special instructions for fragments and injection can be freely lifted to the outer scope. Indeed, the fragment closure bound in $\mathbf{f_0}$ can only reference client elements. Since the server X can only introduce server-side variables, the body of the fragment closure is independent from the server-side code. A similar remark can be made about the client functor F: the functor argument must be on the client, hence it cannot introduce new server binding. The server identifier that is

injected must have been introduced outside of functor and the injection can be lifted outside the functor.

Using this remark, the structure of the ML_S and ML_C programs is fairly straightforward: Code on the appropriate side has the same shape as in the original Eliom$_\varepsilon$ program and code on the other side only contains calls to the appropriate primitives.

Eliom$_\varepsilon$	ML_S	ML_C

module$_s$ X = struct
  let$_s$ a = {{ 2 }}
  let$_s$ b = 4
end;
	module X = struct
	  let a = fragment $\mathbf{f_0}$ ()
	  let b = 4
	end;
		bind $\mathbf{f_0}$ = $\lambda()$.2;

	end ();	exec ();

module$_c$ $F(Y : \mathcal{M})$ = struct
  let$_c$ $a = Y.b$ + int%$X.b$
end;
	injection $\mathbf{x_0}$ (int^s $X.b$);	module $F(Y : \mathcal{M})$ = struct
		  let $a = Y.b$ + (int^c $\mathbf{x_0}$)
		end;

module$_c$ Z =
  F(struct let$_c$ b = 2 end);
		module Z =
		  F(struct let b = 2 end);

let$_c$ return =
  frag%$X.a$ + $Z.a$;
	injection $\mathbf{x_1}$ (frag^s $X.a$);	let return =
		  (frag^c $\mathbf{x_1}$) + $Z.a$;

Example 15. Compilation of client and server modules and functors

Mixed modules. Finally, Example 16 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 compilation of the mixed module X is similar to the procedure for programs: we compile each declaration and use the `injection` primitive as needed. Additionally, we add a `Dyn` field on the server-side version of the module. The content of the `Dyn` field is determined statically for simple structures (here, it is $\mathbf{X_0}$). The client-side version of the module is first bound to $\mathbf{X_0}$ using the bind$_m$ primitive. We then declare X as a simple alias. This alias ensures that X is also usable in client sections transparently.

For functors, the process is similar. One additional complexity is that the `Dyn` field should be dynamically generated. For this purpose, we add a call to the fragment$_m$ primitive. Each call to fragment$_m$ generates a new, fresh identifier. We also prefix each call to `fragment` by the `Dyn` field. On the client, we emit two different functors. The first one is called F and contains only the client declarations to be used inside the rest of the client code. It is used for client-side usage of mixed functors. An example with the interpreted semantics was presented in Section 3.6.3. The other one is bound to a new reference (here $\mathbf{F_1}$) and contains both client declaration, along with calls to the `bind` and `exec` primitives. This function 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).

5.3 Slicing rules

Given an Eliom$_\varepsilon$ module M (resp. module type $\mathcal{M}$, structure S, …) 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.

Let us defines a few notation. As before, we use $e[a \mapsto b]$ 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 $(e_i)_i$ the sequence of elements e_i. For ease of presentation, we use D_ς (resp. $\mathcal{D}_\varsigma$) for definitions (resp. declarations) located on location ς. In order to simplify our presentation of mixed functors, both in the slicing rules and in the simulation proofs, we consider a sliceability constraint which dictates which programs can be sliced.

Definition 1 (Sliceability)**.** A program is said sliceable if mixed structures are only defined at top level, or directly inside a toplevel mixed functor.

We give an example of this constraint in Example 17. The program presented on the left is not sliceable, since it contains a structure which is nested inside a structure in a functor. The semantically equivalent program on the right is sliceable, since structures are not nested. This restriction can be relaxed by using a transformation similar to lambda-lifting on mixed functors. In the rest of this section, we assume programs are sliceable and well typed.

We now describe how to slice the various constructions of our language. The slicing rules for modules and expressions are defined in Figure 39. The slicing rules for structures and declarations are presented in Figure 40.

Base structure and signature components are left untouched. Indeed, according to Proposition 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_ε 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 5.1. In client structure components, we need to handle injections. We associate each injection to a

ELIOM_ε	ML_S	ML_C

`module`$_m$ X = `struct`
  `let`$_s$ $a = 2$
  `let`$_c$ $b = 4 +$ `int%`a
`end;`
	`module` X = `struct`
	  `module Dyn` = $\mathbf{X_0}$;
	  `let` $a = 2$; `end ();`
	  `injection` $\mathbf{x_0}$ (`int`^{s} a);
	`end;`
		`bind`$_m$ $\mathbf{X_0}$ = `struct`
		  `exec ();`
		  `let` $b = 4 +$ (`int`^{c} $\mathbf{x_0}$)
		`end;`
		`module` $X = \mathbf{X_0}$;

`module`$_m$ $F(Y : \mathcal{M})$ = `struct`
  `let`$_s$ $c = \{\{ Y.b \}\}$
  `let`$_c$ $d = 2 * Y.b$
`end;`
	`module` $F(Y : \mathcal{M})$ = `struct`
	  `module Dyn` =
	  `fragment`$_m$ $\mathbf{F_1}$ (Y.Dyn);
	  `let` c = `fragment Dyn.`$\mathbf{f_0}$ ();
	  `end ();`
	`end;`
		`bind`$_m$ $\mathbf{F_1}(Y : \mathcal{M})$ = `struct`
		  `bind`$_m$ `Dyn.`$\mathbf{f}_0 = \lambda().(Y.b)$;
		  `exec ();`
		  `let` $d = 2 * Y.b$
		`end;`
		`module` $F(Y : \mathcal{M})$ = `struct`
		  `let` $d = 2 * Y.b$
		`end;`

`module`$_m$ $Z = F(X)$;	`module` $Z = F(X)$; `end ();`	`module` $Z = F(X)$; `exec ()`

`let`$_c$ `return` =
  `frag%`$Z.c + Z.d$;
	`injection` $\mathbf{x_1}$ (`frag`^{s} $Z.c$);	`let return` =
		  (`frag`^{c} $\mathbf{x_1}$) + $Z.d$;

Example 16. Compilation of a mixed functor

```
module_m F(X : M) = struct
  module_m Y = struct
    ...
  end
end
```

(a) An unsliceable functor

```
module_m Y'(X : M) = struct
  ...
end
module_m F(X : M) = struct
  module_m Y = Y'(X)
end
```

(b) A sliceable functor

Example 17. The sliceability constraint

Type expressions

$$\langle\{\tau\}\rangle_s = \texttt{frag} \qquad \langle\tau_s \rightsquigarrow \tau_c\rangle_s = \langle\tau_s\rangle_s \to \texttt{serial}$$

Signature components

$$\langle\mathcal{D}_b;\mathcal{S}\rangle_\iota = \mathcal{D}_b;\langle\mathcal{S}\rangle_\iota \qquad \left\langle\mathcal{D}_\varsigma;\mathcal{S}\right\rangle_\iota = \begin{cases}\langle\mathcal{S}\rangle_\iota & \text{when } \varsigma \not\succ \iota \\ \left\langle\mathcal{D}_\varsigma\right\rangle_\iota;\langle\mathcal{S}\rangle_\iota & \text{when } \varsigma \succ \iota\end{cases}$$

Declarations and Definitions

$$\begin{aligned}\langle\texttt{type}_\iota\ t_i\rangle_\iota &= \texttt{type}\ t_i & \langle\texttt{type}_\iota\ t_i = \tau\rangle_\iota &= \texttt{type}\ t_i = \langle\tau\rangle_\iota \\ \langle\texttt{val}_\iota\ x_i : \tau\rangle_\iota &= \texttt{val}\ x_i : \langle\tau\rangle_\iota & \left\langle\texttt{module}_\varsigma\ X_i : \mathcal{M}\right\rangle_\iota &= \texttt{module}\ X_i : \langle\mathcal{M}\rangle_\iota \\ \langle\texttt{let}_\iota\ x_i = e\rangle_\iota &= \texttt{let}\ x_i = e & \langle\texttt{module}_\iota\ X_i = M\rangle_\iota &= \texttt{module}\ X_i = \langle M\rangle_\iota\end{aligned}$$

Module Expressions

$$\begin{aligned}\langle\texttt{struct}\ S\ \texttt{end}\rangle_\iota &= \texttt{struct}\ \langle S\rangle_\iota\ \texttt{end} \\ \langle M(M')\rangle_\iota &= \langle M\rangle_\iota\,(\langle M'\rangle_\iota) \\ \langle\texttt{functor}(X_i : \mathcal{M})M\rangle_\iota &= \texttt{functor}(X_i : \langle\mathcal{M}\rangle_\iota)\,\langle M\rangle_\iota \\ \langle\texttt{functor}_m(X_i : \mathcal{M})M\rangle_\iota &= \texttt{functor}(X_i : \langle\mathcal{M}\rangle_\iota)\,\langle M\rangle_\iota\end{aligned}$$

Module Type Expressions

$$\begin{aligned}\langle\texttt{sig}\ S\ \texttt{end}\rangle_\iota &= \texttt{sig}\ \langle S\rangle_\iota\ \texttt{end} \\ \langle\texttt{functor}(X_i : \mathcal{M})\mathcal{M}'\rangle_\iota &= \texttt{functor}(X_i : \langle\mathcal{M}\rangle_\iota)\,\langle\mathcal{M}'\rangle_\iota \\ \langle\texttt{functor}_m(X_i : \mathcal{M})\mathcal{M}'\rangle_\iota &= \texttt{functor}(X_i : \langle\mathcal{M}\rangle_\iota)\,\langle\mathcal{M}'\rangle_\iota\end{aligned}$$

Fig. 39. Slicing – $\langle\cdot\rangle_\iota$

new fresh reference noted **x**. In $\mathrm{ML_s}$, we use the `injection` primitive to register the fact that the given server value should be associated to a given reference. In $\mathrm{ML_c}$, 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 structure components, we apply a similar process to handle fragments. For each fragment, we introduce a reference noted **f**. In $\mathrm{ML_s}$, 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 $\mathrm{ML_c}$, we use the `bind` primitives to associate to each reference a closure where all the escaped values are abstracted.

$$\langle D_b; S\rangle_\iota = D_b; \langle S\rangle_\iota \qquad \langle D_m; S\rangle_\iota = \langle D_m\rangle_\iota; \langle S\rangle_\iota$$

$$\langle D_c; S\rangle_s = \left(\texttt{injection}\ \mathbf{x_i}\ (f_i^s\ x_i);\right)_i \langle S\rangle_s$$

$$\langle D_c; S\rangle_c = \begin{array}{l} D_c\left[\overline{f_i\%x_i} \mapsto \overline{(f_i^c\ \mathbf{x}_i)}\right]; \\ \langle S\rangle_c \end{array} \quad \text{Where} \begin{cases} \overline{f_i\%x_i} = \text{INJS}(D_c) \\ \overline{\mathbf{x}_i} \text{ is a list of fresh variables.} \end{cases}$$

$$\langle D_s; S\rangle_s = \begin{array}{l} \langle D_s\rangle_s\left[\{\{\ e_i\ \}\}_{\mathbf{f_i}} \mapsto \texttt{fragment}\ \mathbf{f_i}\ \overline{(f_{i,j}^s\ a_{i,j})}\right]_i; \\ \texttt{end}\ (); \\ \langle S\rangle_s \end{array}$$

$$\langle D_s; S\rangle_c = \begin{array}{l} \Big(\texttt{bind Dyn.}\mathbf{f_i} = \lambda\overline{x_{i,j}}.e_i\left[f_{i,j}\%a_{i,j} \mapsto (f_{i,j}^c\ x_{i,j})\right];\Big)_i \\ \texttt{exec}\ (); \\ \langle S\rangle_c \end{array} \quad \text{Where} \begin{cases} \overline{\{\{\ e_i\ \}\}_{\mathbf{f_i}}} = \text{FRAGS}(D_s) \\ \forall i, \overline{f_{i,j}\%a_{i,j}} = \text{INJS}(e_i) \\ \forall i, \overline{x_{i,j}} \text{ are fresh variables} \end{cases}$$

$$\left\langle \begin{array}{l} \texttt{module}_m\ F_i\overline{(X_{i_k} : \mathcal{M}_k)} = \texttt{struct} \\ \quad S \\ \texttt{end}_\mathbf{F} \end{array} \right\rangle_s = \begin{array}{l} \texttt{module}\ F_i\overline{(X_{i_k} : \langle\mathcal{M}_k\rangle_s)} = \texttt{struct} \\ \quad \texttt{module Dyn} = \texttt{fragment}_m\ \mathbf{F}\ \overline{(X_{i_k}.\texttt{Dyn})}; \\ \quad \langle S\rangle_s \\ \texttt{end} \end{array}$$

$$\left\langle \begin{array}{l} \texttt{module}_m\ F_i\overline{(X_{i_k} : \mathcal{M}_k)} = \texttt{struct} \\ \quad S \\ \texttt{end}_\mathbf{F} \end{array} \right\rangle_c = \begin{array}{l} \texttt{bind}_m\ \mathbf{F}\overline{(X_{i_k} : \langle\mathcal{M}_k\rangle_c)} = \texttt{struct}\ \langle S\rangle_c\ \texttt{end}; \\ \texttt{module}\ F_i\overline{(X_{i_k} : \langle\mathcal{M}_k\rangle_c)} = \texttt{struct}\ S|_c\ \texttt{end}; \end{array}$$

$$\langle \texttt{module}_m\ X_i = \texttt{struct}\ S\ \texttt{end}_\mathbf{X}\rangle_s = \begin{array}{l} \texttt{module}\ X_i = \texttt{struct} \\ \quad \texttt{module Dyn} = \mathbf{X}; \\ \quad \langle S\rangle_s \\ \texttt{end}_\mathbf{F} \end{array}$$

$$\langle \texttt{module}_m\ X_i = \texttt{struct}\ S\ \texttt{end}_\mathbf{X}\rangle_c = \begin{array}{l} \texttt{bind}_m\ \mathbf{X} = \texttt{struct}\ \langle S\rangle_c\ \texttt{end}; \\ \texttt{module}\ X_i = \mathbf{X}; \end{array}$$

$$\langle \texttt{module}\ X_i = M\rangle_s = \begin{array}{l} \texttt{module}\ X_i = \langle M\rangle_s; \\ \texttt{end}\ (); \end{array}$$

$$\langle \texttt{module}\ X_i = M\rangle_c = \begin{array}{l} \texttt{module}\ X_i = \langle M\rangle_c; \\ \texttt{exec}\ (); \end{array}$$

Fig. 40. Slicing of declarations – $\langle D\rangle_\iota$

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. For each structure, we use the reference annotated on structures, as described in Section 3.6.2. 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. In ML_S,

we add a `Dyn` field to the structure. The value of this field is the result of a call to the primitive $\texttt{fragment}_m$ 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 $\texttt{bind}_m$ primitive

5.4 Typing preservation

One desirable property is that the introduction of new elements in the language and the compilation operation does not compromise the guarantees provided by the host language. To ensure this, we show that slicing a well typed Eliom$_\varepsilon$ program provides two well typed ML_c and ML_s programs.

We only consider typing environments Γ containing the primitive types `frag` and `serial` *i.e.*, $(\texttt{type}_s\ \texttt{frag}) \in \Gamma$ and $(\texttt{type}_b\ \texttt{serial}) \in \Gamma$. We also extend the slicing operation to typing environments. Slicing a typing environment is equivalent to slicing a signature with additional rules for converters. Converters, in Eliom$_\varepsilon$, 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:

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

Theorem 3 (Compilation preserves typing)**.** Let us consider M and $\mathcal{M}$ such that $\Gamma \blacktriangleright_m M : \mathcal{M}$. Then $\langle\Gamma\rangle_s \blacktriangleright \langle M\rangle_s : \langle\mathcal{M}\rangle_s$ and $\langle\Gamma\rangle_c \blacktriangleright \langle M\rangle_c : \langle\mathcal{M}\rangle_c$

Proof. We proceed by induction over the proof tree of $\Gamma \blacktriangleright_m M : \mathcal{M}$. 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 $\Gamma \blacktriangleright_m (D_c; S) : \mathcal{S}$ and $\text{INJS}(D_c) = f\%x$. We note **x** the fresh reference. By definition of the typing relation on Eliom$_\varepsilon$, there exists Γ' and τ_c, τ_s such that $\Gamma \subset \Gamma'$, $\Gamma' \triangleright_s f : \tau_\text{s} \rightsquigarrow \tau_\text{c}$ and $\Gamma' \triangleright_s x : \tau_\text{s}$. We observe that there cannot be any server bindings in D_c, Hence we can assume $\Gamma' = \Gamma$. This is illustrated on the proof tree below.

$$\dfrac{\dfrac{\Gamma \triangleright_s f : \tau_\text{s} \rightsquigarrow \tau_\text{c} \quad \overline{\Gamma \triangleright_s x : \tau_\text{s}}}{\Gamma \triangleright_c f\%x : \tau_\text{c}} \\ \vdots}{\Gamma \blacktriangleright_m (D_c; S) : \mathcal{S}}$$

By definition of slicing on typing environments, $(\texttt{val}\ f^s : \langle\tau_\text{s}\rangle_\text{s} \rightarrow \texttt{serial}) \in \langle\Gamma\rangle_s$ and $(\texttt{val}\ f^c : \texttt{serial} \rightarrow \tau_\text{c}) \in \langle\Gamma\rangle_c$. By definition of ML_c and ML_s typing rules, we have $\langle\Gamma\rangle_s \triangleright_{\text{ML}_\text{s}} (f^s\ x) : \texttt{serial}$ and $\langle\Gamma\rangle_c \triangleright_{\text{ML}_\text{c}} (f^c\ \mathbf{x}) : \tau_\text{c}$.

We easily have that $\langle\Gamma\rangle_s \blacktriangleright_{\text{ML}_\text{s}} \texttt{injection}\ \mathbf{x}\ (f^s\ x) : \varepsilon$, as seen on the proof tree below.

$$\dfrac{\dfrac{(\texttt{val}\ f^s : \langle\tau_\text{s}\rangle_\text{s} \rightarrow \texttt{serial}) \in \langle\Gamma\rangle_s}{\langle\Gamma\rangle_s \triangleright_{\text{ML}_\text{s}} f^c : \langle\tau_\text{s}\rangle_\text{s} \rightsquigarrow \texttt{serial}} \qquad \dfrac{}{\langle\Gamma\rangle_s \triangleright_{\text{ML}_\text{s}} x : \langle\tau_\text{s}\rangle_\text{s}}}{\langle\Gamma\rangle_s \blacktriangleright_{\text{ML}_\text{s}} \texttt{injection}\ \mathbf{x}\ (f^s\ x) : \varepsilon}$$

By induction hypothesis on $\Gamma, (\texttt{val}_c\ x_j : \tau_c) \triangleright_m d_c\left[f\%x \mapsto x_j\right] : \varepsilon$ where v_j is fresh, we have $\langle\Gamma\rangle_c, (\texttt{val}\ x_j : \tau_c) \triangleright_{\text{ML}_\text{c}} d_c\left[f\%x \mapsto x_j\right] : \varepsilon$. We can then replace the proof tree of v_j by the one of

$(f^c\ \mathbf{x})$. 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 $\langle\Gamma\rangle_c \blacktriangleright_{\mathrm{ML_c}} d_c[f\%x \mapsto (f^c\ \mathbf{x})] : \varepsilon$ which allows us to conclude.

$$\dfrac{\dfrac{\dfrac{(\texttt{val}\ f^c : \texttt{serial} \to \tau_c) \in \langle\Gamma\rangle_c}{\langle\Gamma\rangle_c \triangleright_{\mathrm{ML_c}} f^c : \texttt{serial} \rightsquigarrow \tau_c} \qquad \dfrac{}{\langle\Gamma\rangle_c \triangleright_{\mathrm{ML_c}} \mathbf{x} : \texttt{serial}}}{\langle\Gamma\rangle_c \triangleright_{\mathrm{ML_c}} (f^c\ \mathbf{x}) : \tau_c}}{\vdots}}{\langle\Gamma\rangle_c \blacktriangleright_{\mathrm{ML_c}} D_c[f\%x \mapsto (f^c\ \mathbf{x})]\,; S : \mathcal{S}}$$

□

5.5 Semantics preservation

We now state that the compilation process preserves the semantics of Eliom$_\varepsilon$ programs. In order to do that, we show that, given an Eliom$_\varepsilon$ program P, the trace of its execution is the same as the concatenation of the traces of $\langle P\rangle_s$ and $\langle P\rangle_c$.

First, let us put some constraints on the constants of the Eliom$_\varepsilon$, ML$_\mathrm{s}$ and ML$_\mathrm{c}$ language:

Hypothesis 1 (Well-behaved converters)**.** Converters are said to be well-behaved if for each constant c in *Const* such that $\mathit{TypeOf}(c) = \tau_s \rightsquigarrow \tau_c$, then $c^s \in \mathit{Const}_s$ and $c^c \in \mathit{Const}_c$.

We now assume that converters in Eliom$_\varepsilon$, ML$_\mathrm{s}$ and ML$_\mathrm{c}$ are *well-behaved*. We can then state the following theorem.

Theorem 4 (Compilation preserves semantics)**.** Given sets of constants where converters are well-behaved, given an Eliom$_\varepsilon$ program P respecting the slicability hypothesis and such that $P \overset{\{\}}{\Longrightarrow} v, \theta$ then

$$\langle P\rangle_s \overset{\{\}}{\Longrightarrow}_{\mathrm{ML_s}} (), \xi, \zeta, \theta_s \qquad \langle P\rangle_c, \xi \overset{\{\} \mid \zeta \to \gamma}{\Longrightarrow}_{\mathrm{ML_c}} v, \xi', \theta_c \qquad \theta = \theta_s \mathbin{@} \theta_c$$

Proof. The complete proof is given in Appendix B. The proof proceed in the following way: First, we simplify the problem by applying a code transformation that hoist injections to the top level. We then proceed with a proof by simulation for client code (Lemma 1), server code (Lemma 2) and finally mixed code (Lemma 3). □

This theorem not only show that the return value is the same, but also that the trace is identical. This means that side effects happen in the same order and in the same location as specified by the interpreted semantics. While our simplified calculus does not have side effects, OCaml (and thus Eliom) do, making such guarantee essential.

6 STATE OF THE ART AND COMPARISON

Eliom takes inspiration from many sources. The two main influences are, naturally, the extremely diverse ecosystem of web programming languages and frameworks, which we explore in Section 6.1, and the long lineage of ML programming languages . One of the important contributions of Eliom is the use of a programming model similar to languages for distributed systems (Section 6.2) while using an execution model inspired by staged meta-programming (Section 6.3).

6.1 Web programming

Various directions have been explored to simplify Web development and to adapt it to current needs. Eliom places itself in one of these directions, which is to use the same language on the server and the client. Several unified client-server languages have been proposed. They can be

split in two categories depending on their usage of JavaScript. JavaScript can either be used on the server, with Node.js, or as a compilation target, for example with Google Web Toolkit for Java or Emscripten for C. The approach of compiling to JavaScript was also used to develop new client languages aiming to address the shortcomings of JavaScript. Some of them are new languages, such as Haxe, Elm or Dart. Others are only JavaScript extensions, such as TypeScript or CoffeeScript.[3]

However, these proposals only address the fact that JavaScript is an inadequate language for Web programming. They do not address the fact that the model of Web programming itself – server and client aspects of web applications are split in two distinct programs with untyped communication – raises usability, correctness and efficiency issues. A first attempt at tackling these concerns is to specify the communication between client and server. Such examples includes SOAP[4] (mostly used for RPCs) and REST[5] (for Web APIs). A more recent attempt is the GraphQL [GraphQL 2016] query language which attempts to describe, with a type system, the communications between the client and server parts of the application. These proposals are very powerful and convenient ways to check and document Web-based APIs. However, while making the contract between the client and the server more explicit, they further separate web applications into distinct tiers.

Tierless languages attempt to go in the opposite direction: by removing tiers and allowing web applications to be expressed in one single program, they make the development process easier and restore the possibility of encapsulation and abstraction without compromising correctness. In the remainder of this section, we attempt to give a fairly exhaustive taxonomy of tierless programming languages. We first give a high-level overview of the various trade-offs involved, then we give a detailed description of each language.

6.1.1 Code and data location. In Eliom, code and data locations are specified through syntactic annotations. Other approaches for determining locations have been proposed. The first approach is to infer locations based on known elements through a control flow analysis (Stip.js, Opa, Ur/Web): database access is on the server, dynamic DOM interaction is done on the client, etc. Another approach is to extend the type system with locations information (Links, ML5). Locations can then be determined by relying on simple type inference and checking.

These various approaches present a different set of compromises:

- We believe that the semantics of a language should be easy to predict by looking at the code, which is why Eliom uses syntactic annotations to specify locations. This fits well within the OCaml language, which is specifically designed to have a predictable behavior. On the other end of the spectrum, languages with inferred location sacrifice predictability for a very light-weight syntax which provides very little disruption over the rest of the program. Typed based approaches sit somewhere in the middle: locations are not visible in the code but are still accessible through types. Such approaches benefit greatly from IDEs allowing exploring inferred types interactively.
- Naturally, explicit approaches are usually more expressive than implicit approaches. Specifying locations manually gives programmers greater control over the performance of their applications. Furthermore, it allows to express mixed *data structures*, *i.e.*, data structures that contain both server and client parts as presented in Section 2.3. Such idioms are difficult to express when code locations are inferred. We demonstrate this with an example in the Ur/Web description.

[3] A fairly exhaustive list of languages compiling to JavaScript can be found in https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS

[4] https://en.wikipedia.org/wiki/SOAP

[5] https://en.wikipedia.org/wiki/Representational_state_transfer

- Type-directed approaches to infer code location is an extremely elegant approach. It can be employed either in an algebraic effect setting (Links) or as modal logic annotations (ML5). By its type-directed nature, error messages can be expressed in term of the source language and it should lend itself naturally to separate compilation (although this has not yet been achieved). However, such novel type systems significantly extend traditional general purpose type systems (the ML one, in this case) to the point where it seems difficult to retrofit them on an existing languages. One lead would be to provide a tight integration using a form of Foreign Function Interface. Such integration has yet to be proposed.
- Eliom, as a language based on OCaml, is an effectful language. Marrying inference of locations and a side-effecting semantics is delicate. The Stip.js library [Philips et al. 2014] attempts to solve this by automatically providing replication and eventual consistency on shared references. This might cause many more communications that necessary if not done carefully. We believe that such decision is better left in the hands of the programmer.

6.1.2 Slicing. Once code location has been determined, the tierless program must be sliced in (at least) two components. In Eliom, slicing is done statically at compile time in a modular manner: each module is sliced independently. Another common approach is to use a static whole-program slicing transformation (Ur/Web, Stip.js). This is most common for languages where code location is inferred, simply due to the fact that such inference is often non-modular. This allows precise analysis of location that can benefit from useful code transformations such as CPS transformation [Philips et al. 2016], inlining and defunctionalization. However, this can make it difficult for the users to know where each piece of code is executed and hinder error messages,. It also prevents any form of separate compilation.

Finally, slicing can be done at runtime simply by generating client JavaScript code "on the fly" during server execution (Links, Hop, php). Such solution has several advantages: it is easier to implement and provides a very flexible programming style by allowing programmers to compose the client program in arbitrary ways. The downside is that it provides less guarantees to the users. Furthermore, it prevents generating and optimizing a single JavaScript file in advance, which is beneficial for caching and execution purposes.

Separate and incremental compilation. Most current mainstream compiled language support some form of incremental compilation. Indeed, incremental compilation avoids recompiling files of which no dependency has changed. This accelerates the feedback loop between development and testing greatly and allow very fast recompilation times. In the case of statically typed languages, it also allows immediate checking of the modified file thus providing developers very fast iteration cycles. The easiest way to implement incremental compilation is through separate compilation, where each file can be compiled completely independently. Furthermore, separate compilation is compatible with link-time optimization and thus does not prevent generation of heavily optimized code, as demonstrated by nearly every C compiler. As a consequence, we consider languages that do not support incremental compilation completely unusable for practical usages.

6.1.3 Communications. Eliom uses asymmetric communication between client and server (see Section 2.2.5): everything needed to execute the client code is sent during the initial communication that also sends the Web page. It also exposes a convenient API for symmetric communications using RPC (Section 2.3.1) and broadcasts, which must be used manually.

We thus distinguish several kind of communications. First, manual communications are exposed through normal APIs and are performed explicitly by programmers. Of course, the convenience and safety of such functions vary a lot depending on the framework. Then, we consider automatic

communications, that are inserted automatically by the language at appropriate points, as determined by code locations and slicing. We can further decompose automatic communications further in two categories. In static asymmetric communications, information is sent from the server to the client automatically, when sending the page. In dynamic symmetric communications, information is sent back and forth between the client and the server dynamically through some form of channel (AJAX, websockets, Comet, ...).

While symmetric communications are very expressive, they impose a significant efficiency overhead: a permanent connection must be established and each communication imposes a round trip between client and server. Furthermore, such communication channel must be very reliable. On the other hand, asymmetric communications are virtually free: data is sent with the web page (and is usually much smaller). Only a thin instrumentation is needed. Of course, the various communication methods can be mixed in arbitrary manner. Eliom, for example, uses both automatic asymmetric and manual communications.

Offline usage. Many web applications are also used on Mobile phones, where connection is intermittent at best. As such, we must consider the case where the web application produced by a tierless language is used offline. In this context, asymmetric communication offer a significant advantage: given the initially transmitted information by the server, the client program can run perfectly fine without connection. This guarantee, however, does not extend to dynamic manual communications done by the use of RPCs and channels. Philips et al. [2014] explore this question for symmetric communications through their R5 requirement.

6.1.4 Type systems. Type safety in the context of tierless languages can encompass several notions. The first notion is the traditional distinction between weakly and strongly typed languages. In the interest of avoiding a troll war among the jury, we will not comment further. A more interesting question is whether communication errors between client and server are caught by the typechecker. This is, surprisingly, not the case of Ur/Web since location inference and slicing is done very late in the compilation process, far after type checking. One consequence of this is that slicing errors are fairly difficult to understand [Chlipala 2015b, page 10].[6] While the Eliom formalization is type safe, the Eliom implementation is not, due to the use of wrapping and Marshall, which will fail at runtime on functional values.

Another remark is the distinction between client and server universes.[7] Eliom has separate type universes for client and server types (see Section 3.3.3). Most tierless languages do not provide such distinction, notably for the purpose of convenience. Distributed systems such as Acute, however, do make such distinction to provide a solution for API versionning and dynamic reloading of code. In this case, there are numerous distinct type universes.

Module systems. The notion of module system varies significantly depending on the language. In Eliom we consider an ML-style module system composed of a small typed language with structures and functors. We believe modules are essential for building medium to large sized programs: this has been demonstrated for general purpose languages but also holds for web programming languages, as demonstrated by the size of large modern websites (the web frontend of facebook alone is over 9 *millions* lines of code). Even JavaScript recently obtained a module system in ES6. In the context of tierless languages, an interesting question is the interaction between locations and modules. In particular, can modules contain elements of different locations and, for statically typed languages, are locations reflected in signatures?

[6] "However, the approach we adopted instead, with ad-hoc static analysis on whole programs at compile time, leads to error messages that confuse even experienced Ur/Web programmers."

[7] Or, more philosophically: Is your favorite language platonist or nominalist ?

Types and documentation. Type systems are indisputably very useful for correctness purposes, but they also serve significant documentation purposes. Indeed, given a function, its type signature provides many properties. In traditional languages, this can range from very loose (arguments and return types) to very precise (with dependent types and parametricity [Wadler 1989]). In the context of tierless languages, important questions we might want to consider are "Where can I call this function?" and "Where should this argument come from?". The various languages exposes this information in different ways: Eliom does not expose location in the types, but it is present in the signature. ML5 exposes this information directly in the types. Ur/Web and Links do not expose that information at all.

6.1.5 Details on some specific approaches. We now provide an in-depth comparison with the most relevant approaches. A summary in Figure 41 classifies each approach according to the main distinctive features described in the previous paragraphs. Each language or framework is also described below.

	Locations	Slicing	Communications	Type safe	Host language
Eliom	Syntactic	Modular	Asymmetric	✓	OCaml
Links	Type-based*	Dynamic*	Symmetric	✓	-
Ur/Web	Inferred	Global	(A)symmetric~	✓*	-
Haste	Type-based	Modular	Symmetric	✓	Haskell
Hop	Syntactic	Dynamic*	(A)symmetric~	×	JavaScript*
Meteor.js	Syntactic	Dynamic	Manual	×	JavaScript
Stip.js	Inferred	Global	Symmetric*	×	JavaScript
ML5	Type-based	Global*	Symmetric	✓	-
Acute	Syntactic	Modular	Distributed	✓	OCaml

Fig. 41. Summary of the various tierless languages
See previous sections for a description of each headline. A star * indicates that details are available in the description of the associated language. A tilde ~ indicates that we are unsure, either because the information was not specified, or because we simply missed it.

Ur/Web [Chlipala 2015a,b] is a new statically typed language especially designed for Web programming. It features a rich ML-like type and module system and a fairly original execution model where programs only execute as part of a web-server request and do not have any state (the language is completely pure). While similar in scope to Eliom, it follows a very different approach: Location inference and slicing are done through a whole-program transformation operated on a fairly low level representation. Notably, this transformation relies on inlining and removal of high-order functions (which are not supported by the runtime). The advantages of this approach are twofold: It makes Ur/Web applications extremely fast (in particular because it doesn't use a GC: memory is trashed after each request) and it requires very little syntactic overheads, allowing programs to be written in a very elegant manner.

The downsides, however, are fairly significant. Ur/Web's approach is incompatible with any form of separate compilation. Many constructs are hard-coded into the language, such as RPCs and reactive signals and it does not seem possible to implement them as libraries. The language is clearly not general and has a limited expressivity, in particular when trying to use mixed data-structures (see Section 2.3). For example, Example 18. presents the server function `button_list` which takes a list of labels and client functions and generates a list of buttons. We show the Eliom implementation and a tentative Ur/Web implementation. The Ur/Web version typechecks but

slicing fails. We are unable to write a working version and do not believe it to be possible: indeed, in the Eliom version we use a client fragment to build the list `l` as a mixed data-structure. This annotation is essential to make the desired semantics explicit. Some examples are expressible only using reactive signals, which present a very different semantics.

```
let%client handler _ = alert "clicked!"
let%server l =
  [ ("Click!", [%client handler]) ]

let%server button_list lst =
  ul (List.map (fun (name, action) ->
    li [button
      ~button_type : Button
      ~a:[a_onclick action]
      [pcdata name]])
    lst)

let main () =
  body (button_list l)
```

(a) Eliom version

```
fun main () : transaction page =
 let
  fun handler _ = alert "clicked!"
  val l = Cons (("Click!", handler), Nil)

  fun button_list lst =
   case lst of
     Nil => <xml/>
   | Cons ((name, action), r) =>
     <xml>
      <button value={name}
              onclick={action}/>
      {button_list r}
     </xml>
 in
 return <xml>
   <body>{button_list l}</body>
 </xml>
end
```

(b) Tentative Ur/Web version. Typechecks but does not compile.

Example 18. Programs building a list of buttons from a list of client side actions

Hop [Serrano et al. 2006] 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. The implementation of Hop.js is very complete and allow them to run both the JavaScript and the scheme dialect while leveraging the complete node.js ecosystem. Hop uses very similar language constructions to the one provided by Eliom: ~-expressions are fragments and $-expressions are injections. All functions seem to be **`shared`** by default. Communications are asymmetric when possible and use channels otherwise. However, contrary to Eliom, slicing is done dynamically during server execution [Loitsch and Serrano 2007]. In the tradition of Scheme, Hop only uses a minimal type system for optimizations and does not have a notion of location. In particular Hop does not provide static type checking and does not statically enforce the separation of client and server universes (such as preventing the use of database code inside the client). The semantics of Hop has been formalized [Boudol et al. 2012; Serrano and Queinnec 2010] and does present similarities to the interpreted Eliom semantics (Section 3.6). Hop is however significantly more dynamic than Eliom: it allows dynamic communication patterns through the use of channels and allows nested fragments in the style of Lisp quotations which allows to generate client code inside client code.

For dynamically-typed inclined programmers, Hop currently presents the most convincing approach to tierless Web programming. In particular given its solid implementation, great flexibility and support for the JavaScript ecosystem.

Links [Cooper et al. 2006] is an experimental functional language for client-server Web programming with a syntax close to JavaScript and an ML-like type system. Its type system is

extended with a notion of *effects*, allowing a clean integration of database queries in the language [Lindley and Cheney 2012]. In Example 19, we highlight two notable points of Links: the function `adults` takes as argument a list `l` and returns the name of all the person over 18. This function has no effect and can thus run on the client, the server, but can also be transformed into SQL to run in a database query. On the other hand, the `print` function has an effect called "wild" which indicates it can't be run inside a query. Effects are also used to provide type-safe channel-based concurrency.

Links also allows to annotate functions by indicating on which location they should run. Those annotations, however, are not reflected in the type system. Communications are symmetric and completely dynamic through the use of AJAX. Client-server slicing is dynamic (although some progress has been made towards static *query* slicing [Cheney et al. 2014]) and can introduce "code motion", which can moves closures from the server to the client. This can be extremely problematic in practice, both from an efficiency and a security point of view. The current implementation of Links is interpreted but a compilation scheme leveraging the Multicore-OCaml efforts has been recently added.

Although Links is very seducing, the current implementation presents many shortcomings given its statically typed nature: slicing is dynamic and produces fairly large JavaScript code and the type system does not really track client-server locations.

```
1 links> fun adults(l) { for (x <- l) where (x.age >= 18) [(name = x.name)] } ;;
2 adults = fun : ([(age:Int,name:a|_)]) -> [(name:a)]
3
4 links> print ;;
5 print : (String) {wild}-> ()
```

Example 19. Small pieces of Links code

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

Stip.js [Philips et al. 2014] allows to slice tierless JavaScript programs with a minimal amount of annotations. It emits Meteor.js programs with explicit communications. Annotations are optionally provided through the use of comments, which means that Stip.js are actually perfectly valid JavaScript programs. Location inference and slicing are whole-program static transformations. Communications are symmetric, through the use of fairly elaborate consistency and replication mechanisms for shared references. This approach allows the programmer to write code with very little annotations. As opposed to Ur/Web, manual annotations are possible, which might allow to express delicate patterns such as mixed data-structures and prevents security issues.

6.2 Distributed programming

Tierless languages in general are very inspired by distributed programming languages. The main difference being that distributed programs contain an arbitrary number of locations while tierless web programs only have two: client and server. Communications are generally symmetric

and dynamic, due to the multi-headed aspect of distributed systems. There are of course numerous programming languages dedicated to distributed programming. We present here two relevant approaches that put greater emphasis on the typing and tierless aspects.

Haste [Ekblad 2017] is an Haskell EDSL for distributed programming. This DSL allows to express complex orchestrations of multiple nodes and external components (for example databases and IoT components), with handling of distinct type universes when necessary. Instead of using syntactic annotations, locations are determined through typing. This approach works particularly well in the context of Haskell, thanks to the advanced type system and the syntactic support for monads and functors. Multiple binaries are produced from one program. Slicing relies on type information and dead code elimination, as provided by the GHC compiler. Explicit slicing markers similar to Eliom's section annotations are the subject of future work. Communications are dynamic and symmetric through the use of websockets. One notable feature of this DSL is that it offers a client-centric view: The control flow is decided by the client which pilots the other nodes. This is the opposite of Eliom where the server can assemble pieces of client code through fragments. This work also inherits the Haskell and GHC features in term of modules, data abstraction and separate-compilation. A module language has been developed for Haskell by Kilpatrick et al. [2014].

An earlier version, Haste.App [Ekblad and Claessen 2014], was limited to only one client and one server and used a monadic approach to structure tierless programs.

ML5 [Murphy VII et al. 2007] is an ML language that introduces new constructs for type-safe communication between distributed actors through the use of location annotations inside the types called "modal types". It is geared towards a situation where all actors have similar capabilities. It uses dynamic communication, which makes the execution model very different from Eliom. ML5 provides a very rich type system that allows to precisely export the capabilities of the various locations. For example, it is possible to talk about addresses on distant locations and pass them around arbitrary. Eliom only supports such feature through the use of fragments, for client code.

Unfortunately, ML5 does not have a module system. However, we believe that ML5's modal types can play a role similar to Eliom's location annotations on declarations, including location polymorphism. ML5 uses a global transformation for slicing. Given the rich typing information present in ML5's types, it should lend itself fairly well to a modular slicing approach, but this has not been done.

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 message and by versioning APIs. In particular, great care is taken to provide type safe serialization by also transmitting the type of messages alongside each message. This gives Acute very interesting capabilities, such as reloading only part of the distributed system in a type-safe way.

6.3 Staged meta-programming

An important insight regarding Eliom is that, while it is a tierless programming language and tries to disguise itself as a distributed programming language, Eliom corresponds exactly to a staged meta-programming language. Eliom simply provides only two stages: stage 0 is the server, stage 1 is the client. Eliom's client fragments are the equivalent of stage quotations.

Most approaches to partial evaluation are done implicitly (not unlike tierless languages with implicit locations). We take inspiration from several approaches that combine staged meta-programming with explicit stages annotations that are reflected in the type system. We only look at the most relevant approaches but a longer description of the history of staged meta-programming approaches can be found in Taha [1999, Chapter 7].

MetaOCaml [Kiselyov 2014] is an extension of OCaml for meta programming. It introduces a quotation annotation for staged expressions, whose execution is delayed. Quotations and antiquotations corresponds exactly to fragments and injections. The main difference is that MetaOCaml is much more dynamic: quoted code does not have to be completely closed when produced and well-scopedeness is checked dynamically, just before running the quoted code. This allows very dynamic behaviors such as automatic insertion of let-bindings [Kiselyov 2015] and dynamically determining staged stream pipelines [Kiselyov et al. 2017]. One difference is the choice of universes: Eliom has two universes, client and server, which are distinct. MetaOCaml has a single type universe but a series of scopes, for each stage, included in one another.

MetaOCaml itself provides no support for modules and only leverages the OCaml module system. Staging annotations are only on expressions, not on declarations.

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. The slicing can 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. They also use a notion similar to converters, except that the serialization format here is simply the OCaml AST.

The main difference compared to Eliom is how the asymmetry between stage 0 and stage 1 is treated. Only one type universe is used and there is no notion of slicing that would allow a distant execution.

Feltman et al. [2016] presents a slicing technique for a two-staged simply typed lambda calculus. Their technique is similar to the one used in Eliom. They distinguish their language it three parts: $1\mathcal{G}$, which corresponds to base code; $1\mathcal{M}$, which corresponds to server code; and $2\mathcal{M}$, which corresponds to client code. They also provide a proof of equivalence between the dynamic semantics and the slicing techniques. This proof has been mechanized in Twelf. While their work is done in a more general settings, they do not specify how to transfer rich data types across stages (which is solved in Eliom using converters). They also do not propose a module system.

7 CONCLUSION

We presented Eliom, a Modular Tierless Web programming language that supports static typing, an efficient execution model, abstraction and modularity. We showed its design through numerous examples and formalized it. In our formalization, we first gave a simple interpreted semantics, that is suitable to explain to programmers. We then gave a more efficient compilation scheme and showed that it preserves both typing and semantics. The compilation scheme follows precisely the current implementation of Eliom.

Many design choices of Eliom were inspired by practical concerns. Indeed, in order to be useful, a language must have an ecosystem. The simplest way to have an ecosystem is to reuse the one of an existing language, in our case the OCaml one. Since separate compilation and modularity are indispensable for any non-trivial programming projects, we needed to make modularity, abstraction and the tierless annotations interact. Eliom is thus inspired by many different elements such as ML

languages, tierless web programming and staged meta-programming. In particular, Eliom aims to combine a semantics inspired by staged meta-programming and an ML-style module system.

We believe that such combination can be used in a wider context than simply Web programming. Indeed, Ocsigen has also been used to create Mobile applications [Besport 2017]. We would like to extend our language to be usable to any kind of client-server applications that requires dynamicity, type safety and modularity. We also consider extending our model to explicitely support multiple distinct clients. Finally, we aim to further explore the programming patterns made possible by Eliom, for instance multi-tiers functional reactive programming.

REFERENCES

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. 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, 666–679. https://doi.org/10.1145/3009837

Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering formal metatheory. In *Proceeding of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, George C. Necula and Philip Wadler (Eds.). ACM, 3–15. http://arthur.chargueraud.org/research/2007/binders/

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

Besport 2017. *BeSport*. http://www.besport.com/.

Didier Le Botlan and Didier Rémy. 2003. ML^F: raising ML to the power of system F. In *ICFP*. ACM, 27–38.

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. 2014. Effective quotation: relating approaches to language-integrated query. In *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation, PEPM 2014, January 20-21, 2014, San Diego, California, USA*, Wei-Ngan Chin and Jurriaan Hage (Eds.). ACM, 15–26. https://doi.org/10.1145/2543728.2543738

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. https://doi.org/10.1145/2676726.2677004

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

Anton Ekblad. 2017. A meta-EDSL for distributed web applications. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, Iavor S. Diatchki (Ed.). ACM, 75–85. https://doi.org/10.1145/3122955.3122969

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. https://doi.org/10.1145/2633357.2633367

Nicolas Feltman, Carlo Angiuli, Umut A. Acar, and Kayvon Fatahalian. 2016. Automatically Splitting a Two-Stage Lambda Calculus, See [Thiemann 2016], 255–281. https://doi.org/10.1007/978-3-662-49498-1_11

J. Garrigue. 2009. A Certified Interpreter for ML with Structural Polymorphism. (2009).

Gencore 2017. *New York University Gencore*. http://gencore.bio.nyu.edu/.

GraphQL 2016. GraphQL. (2016). http://graphql.org/

Hashids 2017. *Hashids*. http://hashids.org/.

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. https://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. https://doi.org/10.1007/978-3-319-07151-0_6

Oleg Kiselyov. 2015. Generating Code with Polymorphic let: A Ballad of Value Restriction, Copying and Sharing. In *Proceedings ML Family / OCaml Users and Developers workshops, ML Family/OCaml 2015, Vancouver, Canada, 3rd & 4th September 2015. (EPTCS)*, Jeremy Yallop and Damien Doligez (Eds.), Vol. 241. 1–22. https://doi.org/10.4204/EPTCS.241.1

Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. 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, 285–299. http://dl.acm.org/citation.cfm?id=3009880

Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a mechanized metatheory of standard ML. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 173–184. https://doi.org/10.1145/1190216.1190245

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. https://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. https://doi.org/10.1145/199448.199476

Xavier Leroy. 1996. A Syntactic Theory of Type Generativity and Sharing. *J. Funct. Program.* 6, 5 (1996), 667–698. https://doi.org/10.1017/S0956796800001933

Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, Benjamin C. Pierce (Ed.). ACM, 91–102. https://doi.org/10.1145/2103786.2103798

Florian Loitsch and Manuel Serrano. 2007. Hop Client-Side Compilation. In *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007. (Trends in Functional Programming)*, Marco T. Morazán (Ed.), Vol. 8. Intellect, 141–158.

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

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

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/.

Scott Owens. 2008. A Sound Semantics for OCamllight. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science)*, Sophia Drossopoulou (Ed.), Vol. 4960. Springer, 1–15. https://doi.org/10.1007/978-3-540-78739-6_1

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics, See [Thiemann 2016], 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

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. https://doi.org/10.1145/2661136.2661146

Laure Philips, Joeri De Koster, Wolfgang De Meuter, and Coen De Roover. 2016. Dependence-driven delimited CPS transformation for JavaScript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*, Bernd Fischer and Ina Schaefer (Eds.). ACM, 59–69. https://doi.org/10.1145/2993236.2993243

Pumgrana 2017. *Pumgrana.* http://www.pumgrana.com/.

Gabriel Radanne, Vasilis Papavasileiou, Jérôme Vouillon, and Vincent Balat. 2016a. Eliom: tierless Web programming from the ground up. In *IFL 2016, Leuven, Belgium, August 31 - September 2, 2016*, Tom Schrijvers (Ed.). ACM, 8:1–8:12. https://doi.org/10.1145/3064899.3064901

Gabriel Radanne and Jérôme Vouillon. 2018. Tierless Web programming in the large. In *WWW 2018, Lyon, Paris.*

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

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

Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop: a language for programming the Web 2.0. In *OOPSLA Companion*. 975–985.

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. https://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. https://doi.org/10.1017/S0956796807006442

Walid Taha. 1999. *Multi-stage Programming: Its Theory and Applications.* Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology.

Peter Thiemann (Ed.). 2016. *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings.* Lecture Notes in Computer Science, Vol. 9632. Springer. https://doi.org/10.1007/978-3-662-49498-1

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

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

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

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

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

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 347–359. https://doi.org/10.1145/99370.99404

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

A THE ML CALCULUS

This section describes our version of the ML calculus, which contains the minimal amount of ingredients that allows us to describe the ELIOM extensions: a core calculus with polymorphism, let bindings and parametrized datatypes in the style of Wright and Felleisen [1994], accompanied by a fully featured module system with separate compilation and applicative functors in the style of Leroy [1995]. We first present the syntax of the language in Appendix A.1 and its type system in Appendix A.2. We then present the semantics of this language in Appendix A.3. The description in this appendix is self-contained, and thus restate the basic description found in Section 3.1.

A.1 Syntax

Expressions

$e ::= c$ (Constant)
$\mid x_i \mid p.x$ (Variables)
$\mid \mathtt{Y}$ (Fixpoint)
$\mid (e\ e)$ (Application)
$\mid \lambda x.e$ (Function)
$\mid \mathtt{let}\ x = e\ \mathtt{in}\ e$ (Let binding)
$c \in \mathit{Const}$ (Constants)

Path

$p ::= X_i \mid p.X \mid p_1(p_2)$

Type Schemes

$\sigma ::= \forall(\alpha)^*.\tau$

Type Expressions

$\tau ::= \alpha$ (Type variables)
$\mid \tau \rightarrow \tau$ (Function types)
$\mid (\tau^*)\mathrm{t_i} \mid (\tau^*)\mathrm{p.t}$ (Type constructors)

(a) The expression language

Module Expressions

$M ::= X_i \mid p.X$ (Variables)
$\mid (M : \mathcal{M})$ (Type constraint)
$\mid M_1(M_2)$ (Functor application)
$\mid \mathtt{functor}(X_i : \mathcal{M})M$ (Functor)
$\mid \mathtt{struct}\ S\ \mathtt{end}$ (Structure)

Structure body

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

Structure components

$D ::= \mathtt{let}\ x_i = e$ (Values)
$\mid \mathtt{type}\ \mathrm{t_i} = \tau$ (Types)
$\mid \mathtt{module}\ X_i = M$ (Modules)

Programs

$P ::= \mathtt{prog}\ S\ \mathtt{end}$

Module types

$\mathcal{M} ::= \mathtt{sig}\ \mathcal{S}\ \mathtt{end}$ (Signature)
$\mid \mathtt{functor}(X_i : \mathcal{M}_1)\mathcal{M}_2$ (Functor)

Signature body

$\mathcal{S} ::= \varepsilon \mid \mathcal{D}; \mathcal{S}$

Signature components

$\mathcal{D} ::= \mathtt{val}\ x_i : \tau$ (Values)
$\mid \mathtt{type}\ \mathrm{t_i} = \tau$ (Types)
$\mid \mathtt{type}\ \mathrm{t_i}$ (Abstract types)
$\mid \mathtt{module}\ X_i : \mathcal{M}$ (Modules)

Environments

$\Gamma ::= \mathcal{S}$

(b) The module language

Fig. 42. ML grammar

Let us first define some notations and meta-syntactic variables. As a general rule, the expression language is in lowercase (e) and the module language is in uppercase (M). Module types are in calligraphic letters ($\mathcal{M}$). More precisely: x are variables, p are module paths, X are module variables, τ are type expressions and t are type constructors. x_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. Only the name part of identifiers is exposed in module signature. α-conversion should keep the name intact and change only the stamp, which allow to preserve module signatures. Sequences are noted with a star; for example τ^* is a sequence of type expressions. Indexed sequences are noted (τ_i), with an implicit range. Substitution of a by b in e is noted $e[a \mapsto b]$. Repeated substitution of each a_i by the corresponding b_i is noted $e[a_i \mapsto b_i]_i$. The syntax is presented in Figure 42.

Expressions. The expression language is a fairly simple extension of the lambda calculus with a fixpoint combinator `Y` and let bindings `let` x `=` e_1 `in` e_2. The language is parametrized by a set of constants *Const*. Variables can be qualified by a module path p. Paths can be either module identifiers such as X_i, a submodule access such as $X_i.Y$, or a path application such as $X_i(Y_j.Z)$. Note that, as said earlier, that fields of modules are only called by their name, without stamp.

Types. Types are composed of type variables α, function types $\tau_1 \to \tau_2$ and parametrized type constructors $(\tau_1, \tau_2, \ldots, \tau_k)\mathrm{t_i}$. Type constructors can have an arbitrary number of parameters, including zero. Type constructors can be qualified by a module path p. Type schemes, noted σ, are type expressions that are universally quantified by a list of type variables. Type schemes can also have free variables. For example: $\forall\alpha.(\alpha, \beta)\mathrm{t_i}$.

Modules. The module language is quite similar to a simple lambda calculus: Functors are functions over module (except that arguments are annotated with their types). Module application is noted $M_1(M_2)$. Modules can also be constrained by a module type: $(M : \mathcal{M})$. Finally, a module can be a structure which contains a list of value, types or module definitions: `struct let` x_i `= 2 end`. Programs are lists of definitions.

Module types. Module types can be either the type of a functor or a signature, which contains a list of value, types and module descriptions. Type descriptions can expose their definition or can be left abstract. Typing environments are simply module signatures. We note them Γ for convenience.

A.2 Type system

We now present the ML type system. For ease of presentation, we proceed in two steps: we will first forget that the module language exists, and present a self-contained type system for the expression language. We then extend the typing relation to handle modules.

A.2.1 The expression language. We introduce the following judgments:

$\Gamma \triangleright e : \tau$	The expression e has type τ in the environment Γ. See Figure 44.
$\Gamma \triangleright \tau_1 \approx \tau_2$	Types τ_1 and τ_2 are equivalent in environment Γ. See Figure 45.
$\Gamma \models \tau$	The type τ is well formed in the environment Γ. See Figure 43.

We note *TypeOf*(c) the type scheme of a given constant c. The instanciation relation is noted $\sigma \succ \tau$ for a type scheme σ and a type τ. The converse operation which closes a type according to an environment is noted *Close*(Γ, τ). We use $(D) \in \Gamma$ to test if a given type or value is declared in the environment Γ. Note that for types, $(\mathtt{type}\ (\alpha_i)\mathrm{t}) \in \Gamma$ holds also if t is not abstract in Γ.

Polymorphism. One of the main benefit of programming language of the ML family is the ability to easily define and use functions that operate on values of various types. For example, the `map` function can applies to all lists, regardless of the type of their content. Indeed, the type of `map` is polymorphic:

$$\mathtt{map} : \forall\alpha\beta.\ (\alpha \to \beta) \to (\alpha)\mathrm{list} \to (\beta)\mathrm{list}$$

From a type checking point of view, this is possible thanks to two operations: instanciation and abstraction. Instanciation takes a type scheme, which is a type where some variables have been universally quantified, and replace all the quantified type variables by some type. It is used when looking up a variable (rule Var) or typechecking a constant (rule Const). For example, the type of map can be instantiated to the following type.

$$(\text{int} \to \text{bool}) \to (\text{int})\text{list} \to (\text{bool})\text{list}$$

Once instantiated, the `map` function can be applied on a list with concretes types. Naturally, we also need the converse operation: constructing a type scheme given a type containing some type variables. Closing a type depends on the current typing environments, we only abstract type variables that have not been introduced by previous binders. $\mathit{Close}(\Gamma, \tau)$ returns the type scheme $\forall \alpha_1 \ldots \alpha_n.\tau$ where the α_i are free variables of τ that are not present in Γ. While it is possible to apply the closing operation at any step of a typing derivation, it is only useful at the introduction point of type variables, in let bindings (rule LetIn). In the following example, we derive a polymorphic type for a function that constructs a pair with an element from the environment. We first use the close operation to obtain a type scheme for f. Note that since α is present in the environment, it is not universally quantified. We then use the instance operation to apply f to an integer constant.

$$\dfrac{\dfrac{\dfrac{\vdots}{(\texttt{val}\ a : \alpha; \texttt{val}\ b : \beta) \triangleright (a, b) : \alpha * \beta}}{(\texttt{val}\ a : \alpha) \triangleright \lambda b.(a, b) : \beta \to \alpha * \beta} \qquad \dfrac{\dfrac{\forall \beta.\ \beta \to \alpha * \beta > \text{int} \to \alpha * \text{int}}{\Gamma \triangleright f : \text{int} \to \alpha * \text{int}} \qquad \dfrac{\mathit{Const}(3) > \mathit{int}}{\Gamma \triangleright 3 : \text{int}}}{(\texttt{val}\ a : \alpha; \texttt{val}\ f : \forall \beta.\ \beta \to \alpha * \beta) \triangleright f\ 3 : (\alpha * \text{int})}}{(\texttt{val}\ a : \alpha) \triangleright \texttt{let}\ f = \lambda b.(a, b)\ \texttt{in}\ f\ 3 : (\alpha * \text{int})}$$

Parametric datatypes. Parametric polymorphism introduces type variables in type expressions. In the presence of type definitions, it is natural to expect the ability to write type definitions which can contain type variables. This leads us to parametric datatypes: datatypes which are parametrized by a set of variables. (α)list is of course an example of such datatype. Note that care must be taken when deciding the equivalence of types. It the type is not abstract, *i.e.*, its definition is available, we can always unfold the definition, as shown in rule DefTypeEq. However, when considering an abstract type, we cannot unfold the type definition. Instead, we check that head symbols are compatible and that parameters are equivalent pairwise[8] . This is done in rule AbsTypeEq.

$$\textsc{TypeVal}\ \dfrac{(\texttt{type}\ (\alpha_i)\mathrm{t}) \in \Gamma \qquad \forall i,\ \Gamma \vDash \tau_i}{\Gamma \vDash (\tau_i)\mathrm{t}} \qquad \textsc{ArrowVal}\ \dfrac{\Gamma \vDash \tau_1 \qquad \Gamma \vDash \tau_2}{\Gamma \vDash \tau_1 \to \tau_2} \qquad \textsc{VarVal}\ \dfrac{}{\Gamma \vDash \alpha}$$

Fig. 43. Type validity rules – $\Gamma \vDash \tau$

A.2.2 The module language. We introduce the following judgments:

$\Gamma \blacktriangleright M : \mathcal{M}$	The module M is of type $\mathcal{M}$ in Γ. See Figure 46.
$\Gamma \blacktriangleright \mathcal{M} <: \mathcal{M}'$	The module type $\mathcal{M}$ is a subtype of $\mathcal{M}'$ in Γ. See Figure 47.
$\Gamma \vDash \mathcal{M}$	The module type $\mathcal{M}$ is well-formed in Γ. See Figure 48.

The typing rules for OCaml-style modules are quite complex. In particular, the inner details of the rules are not well known, even by OCaml programmers. Before presenting the typing rules

[8]This is similar to the handling of free symbols in the unification literature.

$$\textsc{Var}\ \frac{(\texttt{val}\ x : \sigma) \in \Gamma \qquad \sigma \succ \tau}{\Gamma \triangleright x : \tau} \qquad \textsc{Lambda}\ \frac{\Gamma; (\texttt{val}\ x : \tau_1) \triangleright e : \tau_2}{\Gamma \triangleright \lambda x.e : \tau_1 \to \tau_2} \qquad \textsc{Const}\ \frac{\mathit{TypeOf}(c) \succ \tau}{\Gamma \triangleright c : \tau}$$

$$\textsc{LetIn}\ \frac{\Gamma \triangleright e_1 : \tau_1 \qquad \Gamma; (\texttt{val}\ x : \mathit{Close}(\tau_1, \Gamma)) \triangleright e_2 : \tau_2}{\Gamma \triangleright \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2} \qquad \textsc{Equiv}\ \frac{\Gamma \triangleright e : \tau_1 \qquad \Gamma \triangleright \tau_1 \approx \tau_2}{\Gamma \triangleright e : \tau_2}$$

$$\textsc{App}\ \frac{\Gamma \triangleright e_1 : \tau_1 \to \tau_2 \qquad \Gamma \triangleright e_2 : \tau_1}{\Gamma \triangleright (e_1\ e_2) : \tau_2} \qquad \textsc{Y}\ \frac{}{\Gamma \triangleright \mathsf{Y} : ((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to \tau_1 \to \tau_2}$$

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

Fig. 44. ML expression typing rules – $\Gamma \triangleright e : \tau$

$$\textsc{ReflEq}\ \frac{}{\Gamma \triangleright \tau \approx \tau} \qquad \textsc{TransEq}\ \frac{\Gamma \triangleright \tau_1 \approx \tau_2 \quad \Gamma \triangleright \tau_2 \approx \tau_3}{\Gamma \triangleright \tau_1 \approx \tau_3} \qquad \textsc{CommEq}\ \frac{\Gamma \triangleright \tau_2 \approx \tau_1}{\Gamma \triangleright \tau_1 \approx \tau_2} \qquad \textsc{FunEq}\ \frac{\Gamma \triangleright \tau_1 \approx \tau_1' \quad \Gamma \triangleright \tau_2 \approx \tau_2'}{\Gamma \triangleright \tau_1 \to \tau_2 \approx \tau_1' \to \tau_2'}$$

$$\textsc{DefTypeEq}\ \frac{(\texttt{type}\ (\alpha_i)\mathsf{t} = \tau) \in \Gamma}{\Gamma \triangleright (\tau_i)\mathsf{t} \approx \tau[\alpha_i \mapsto \tau_i]_i} \qquad \textsc{AbsTypeEq}\ \frac{(\texttt{type}\ (\alpha_i)\mathsf{t}) \in \Gamma \quad \forall i,\ \Gamma \triangleright \tau_i \approx \tau_i'}{\Gamma \triangleright (\tau_i)\mathsf{t} \approx (\tau_i')\mathsf{t}}$$

Fig. 45. Type equivalence rules – $\Gamma \triangleright \tau \approx \tau'$

in details, we will attempt to give insight on why some features are present in the languages and what are their advantages. For this purpose, we present two examples illustrating the need for applicative functors and strengthening, respectively. We assume that readers are familiar with simpler usages of ML modules.

Applicative Functors. Let us consider the following scenario: we are given a module G implementing a graph data-structure and would like to implement a simple graph algorithm which takes a vertex and returns all the accessible vertices. We would like the returned module to contain a function of type G.graph → G.vertex → set_of_vertices. How to implement set_of_vertices? An easy but inefficient way would be to use lists. A better way is to use proper sets (implemented with balanced binary tree, for example). In OCaml, this is provided in the standard library by the functor *Set.Make*, presented in Section 2.1, which takes a module implementing comparison functions for the given type. We would obtain a signature similar to the one below.

```
module Access(G : Graph) : sig
  module VerticesSet : sig ... end
  val run : G.graph → G.vertex → VerticesSet.set
end
```

However, this means we need to expose a complete module implementing set of vertices that is independent from any other set module. This prevents modularity, since any usage of our new

function must use this specific set implementation. Furthermore, this make the signature bigger than strictly necessary. What we really want to expose is that the return type comes from an application of *Set.Make*. Fortunately, we can do so by using the following signature.

```
module Access(G : Graph) : sig
  val run : G.graph → G.vertex → Set.Make(G.Vertex).set
end
```

Here, we export the fact that the set type must be the result of a functor application on a module that is compatible with G.Vertex. The type system guarantees that any such functor application will produces types that are equivalent. In particular, if multiple libraries uses the *Access* functor, their sets will be of the same types, which make composition of libraries easier. This behavior of functors is usually called *applicative*.

Strengthening. Let us now consider the program presented in Example 20. We assume the existence of two modules, presented in Example 20a. The module *Showable* exposes the abstract type t, along with a *show* function that turns it into a string. The module *Elt* exposes a type t equal to Showable.t and a value that inhabits this type. The program is presented in Example 20b. We define a functor *F* taking two arguments *E* and *S* whose signature are similar to *Elt* and *Showable*, respectively. The main difference is that *E* comes first and S.t is defined as an alias of E.t. The functor uses the *show* function on the element in *E* to create a string. It is natural to expect the functor application $F(Elt)(Showable)$ to type check, since Elt.t = Showable.t. We must, however, check for module inclusion. While *Elt* is clearly included in the signature of the argument *E*, the same is not clear for *Showable*. We first need to enrich its type signature with additional type equalities. We give *Showable* the type `sig type` t = $Showable.t$ ... `end`. It makes sense to enrich the signature in such a manner since *Showable* is already in the environment. Given this enriched signature, we can now deduce that $\blacktriangleright$(`type` t = $Showable.t$) <: (`type` t = $E.t$) since E.t = Elt.t = Showable.t.

```
module Showable : sig
  type t
  val show : t → string
end
module Elt : sig
  type t = Showable.t
  val v : elt
end
```

(a) Typing environment

```
module F
  (E : sig type t val v : t end)
  (S : sig type t = E.t val show : t → string end)
 = struct
  let s = (S.show E.v)
end
module X = F(Elt)(Showable)
```

(b) Application of multi-argument functor using manifests

Example 20. Program using functors and manifest types

The operation that consists in enriching type signatures of module identifiers with new equalities by using elements in the environment is called *strengthening* [Leroy 1994].

A.2.3 Typing rules. In the previous two examples, we showcased some delicate interactions between functor, type equalities and modularity in the context of an ML module system. We now see in details how the rules presented in Figures 46, 47 and 48 produce these behaviors.

Qualified access. Unqualified module variables are typechecked in a similar manner than regular variables in the expression language, with the ModVar typing rule. Qualified access (of the form $X.a$), both for the core and the module language, need more work. As with the expression language,

the typing environment is simply a list of declaration. In particular, typing environments do not store paths. This means that in order to prove $\Gamma \triangleright p.a : \tau$, we must first verify that the module p typechecks in Γ: $\Gamma \blacktriangleright p : \mathcal{M}$. We then need to verify that the module type $\mathcal{M}$ contains a declaration ($\texttt{val}\ a\ :\ \tau$). This is done in the QualModVar rule for the module language. The rules for the expression language are given in Figure 49.

Let us now try to apply these rules to the module X with the following module type. X contains a type t and a value a of that type. We note that module type $\mathcal{X}$.

$$X : \texttt{sig type } t\texttt{; val } a : t \texttt{ end}$$

We wish to typecheck $X.a$. One expected type for this expression is $X.t$. However, the binding of v in $\mathcal{X}$ gives the type t, with no mention of X. We need to prefix the type variable t by the access path X. This is done in the rule QualModVar by the substitution $\mathcal{M}[n_i \mapsto p.n \mid n_i \in \mathrm{BV}(\mathcal{S}_1)]$ which prefixes all the bound variables of $\mathcal{S}_1$, noted $\mathrm{BV}(\mathcal{S}_1)$, by the path p. Note here that we substitute only by the names declared *before* the variable a. Indeed, a variable or a type can only reference names declared previously in ML. To prove that $X.a$ has the type $X.t$, we can write the following type derivation.

$$\textsc{QualVar}\ \dfrac{\textsc{ModVar}\ \dfrac{(\texttt{module } X : \mathcal{X}) \in (\texttt{module } X : \mathcal{X})}{(\texttt{module } X : \mathcal{X}) \blacktriangleright X : \texttt{sig type } t\texttt{; val } a : t \texttt{ end}}}{(\texttt{module } X : \mathcal{X}) \triangleright X.a : \mathrm{X.t}}\ \text{with } X.t = t[t \mapsto p.t]$$

Strengthening. The strengthening operation, noted $\mathcal{M}/p$, is defined in Figure 50 and is used in the Strength rule. It takes a module type $\mathcal{M}$ and a path p and returns a module type $\mathcal{M}'$ where all the type declarations, abstract or not, have been replaced by type aliases pointing to the path p. These type aliases are usually called "manifest types". This operator relies on the following idea: if p is of type $\mathcal{M}$, then p is available in the environment. In order to expose as many type equalities as possible, it suffices to give p a type where all the type definition point to definitions available in the environment. This way, we preserve type equalities even for abstract types. This also mean that type equalities can be deduced by only looking at the path and the module type. In particular, we do not need to look at the implementation of p, which is important for the purpose of separate compilation.

Applicative functors. Let us consider a functor F with the following type. It takes a module containing a single type t and return a module containing an abstract type t' and a conversion function.

$$F : \texttt{functor}(X : \texttt{sig type } t \texttt{ end})(\texttt{sig type } t'\texttt{; val } make : \mathrm{X.t} \to \mathrm{t}'\ \texttt{end})$$

If we consider two modules X_1 and X_2, does $X_1 = X_2$ imply $F(X_1).t = F(X_2).t$? If that is the case, we say that functors are *applicative*. Otherwise, they are *generative*[9]. Here, we consider the applicative behavior of functors. This is implemented with the last strengthening rule which ensures that the body of functors is also strengthened. For example, if $\mathcal{M}$ is the type of the functor above, $\mathcal{M}/F$ is the following module type:

$$\texttt{functor}(X : \texttt{sig type } t \texttt{ end})(\texttt{sig type } t' = F(X).t\texttt{; val } make : \mathrm{X.t} \to \mathrm{t}'\ \texttt{end})$$

This justifies the presence of application inside paths. Otherwise, such type manifests inside functors could not be represented. A more type-theoretic description of generative and applicative functors can be found in Leroy [1996].

[9]SML only supports generative functors. OCaml originally only supported applicative functors, but also supports the generative behavior since version 4.03.

Separate compilation. Separate compilation is an important properties of programming languages. In fact, almost all so-called "mainstream" languages support it. We can distinguish two aspects of this property: separate typechecking and separate code generation. In both cases, it means that in order to process the file (either to type check it or to transform it into another representation), we only need to look at the type of its dependencies, not their implementation.

It turns out that the ML module system with manifest types lends itself very well to separate typechecking [Leroy 1994]. Indeed, let us consider a program as a list of modules. Each module represents a compilation unit (*i.e.*, a file). Since module bindings in the typing environment only contains module types, and not the actual module, typechecking a file only needs the module type of the previous files, which ensure that we can typecheck each file separately, as long as all its dependencies have been typechecked before. This is expressed more formally in Theorem 5.

Theorem 5 (Separate Typechecking)**.** Given a list of module declarations that form a typed program, there exists an order such that each module can be typechecked with only knowledge of the type of the previous modules.

More formally, given a list of n declarations D_i and a signature $\mathcal{S}$ such that

$$\blacktriangleright (D_1; \ldots; D_n) : \mathcal{S}$$

then there exists n definitions $\mathcal{D}_i$ and a permutation π such that

$$\forall i < n, \mathcal{D}_1; \ldots; \mathcal{D}_i \blacktriangleright D_{i+1} : \mathcal{D}_{i+1} \qquad \blacktriangleright \mathcal{D}_{\pi(1)}; \ldots; \mathcal{D}_{\pi(n)} <: \mathcal{S}$$

Proof. It is always possible to reorder declarations in a signature using the SubStruct rule. This means we can choose the appropriate permutation of definitions that matches the order of declarations. The rest follows by definition of the typing relation. □

A.3 Semantics

We now define the semantics of our ML language. We use a rule-based big step semantics with traces. Traces allows us to reason about execution order in a way that is compatible with modules, as we will see in Appendix A.3.1.

We note v for values in the expression language and V for values in the module language. Values are defined in Figure 52. Values in the expression language can be either constants or lambdas. Module values are either structures, which are list of bindings of values, or functors. We note ρ the execution environment. Execution environments are a list of value bindings. Note here that the execution environment is not mutable, since reference cells are not in the language. We note the concatenation of environment $+$. Environment access is noted $\rho(x) = v$ where x has value v in ρ. The same notation is also used for structures. Traces are lists of messages. For now, we consider messages that are values and are emitted with a `print` operation. The empty trace is noted $\langle\rangle$. Concatenation of traces is noted @.

Given an expression e (resp. a module m), an execution environment ρ, a value v (resp. V) and a trace θ,

$$e \overset{\rho}{\Longrightarrow} v, \theta$$

means that e reduces to v in ρ and prints θ. The reduction rules are given in Figure 53. The rules for the expression language are fairly traditional. Variables and paths must be resolved using the Var and QualVar rules. Applications are done in two steps: first, we reduce both the function and the argument with the App rule, then we apply the appropriate reduction rule for the application: Beta for lambda expressions, Y for fixpoints and Delta for constants. The δ operation gives meaning to application of a constant to a value. $\delta(c, v) = v', \theta$ means that c applied to v returns v' and emits

the trace θ. Let bindings are treated in a similar manner than lambda expressions: the left hand side is executed, added to the environment, then the right hand side is executed.

The module language has similar rules for identifiers and application. In this case, the Beta and App rule have been combined in ModBeta. Additional rules for declarations are also present. Type declarations are ignored (TypeDecl). Values and module declarations (ValDecl and ModDecl) are treated similarly to let bindings: the body of the binding is executed, added to the environment and then the rest of the structure is executed.

A.3.1 Traces and Printing. Traces allow us to visualize the execution order of programs. In particular, if we prove that code transformation preserves traces, it ensures that the execution order is preserved. Traces allow us to reason about execution without introducing references and other side-effecting operations in our language, which would make the presentation significantly more complex.

One example of operation using traces is the `print` constant. Typing and semantics of `print` are provided in Figure 51. `print` accepts any value, prints it, and returns it. From a typing point of view, `print` has the same type as the identity: a polymorphic function which returns its input. We make use of the fact that the Const typing rule also uses the instanciation for type schemes. The semantics of `print` is provided via the Delta rule: it returns its argument directly but also emits a trace containing the given argument.

We now present an example using `print`. We assume the existence of the type `int`, a set of constant corresponding to the integers and an associated operation +. We wish to type and execute the expression e defined as $\texttt{let}\ x = (\texttt{print}\ 3)\ \texttt{in}\ (\texttt{print}\ (x + 1))$

Let us first show that e is of type `int`. The type derivation is provided in Example 21. The typing derivation is fairly direct: we use the Const rule to type `print` as $\texttt{int} \to \texttt{int}$ and apply it to integers with the rule App. We can now look at the execution of e, which returns 4 with a trace $\langle 3; 4\rangle$. The execution derivation is shown in Example 22. The first step is to decompose the let-binding. We first reduce (`print` 3), which can be directly done with the Delta rule. This gives us 3 with a trace $\langle 3\rangle$. We then reduce $(\texttt{print}\ (x + 1))$ in the environment where x is associated to 3. Before resolving the application of `print` with the Delta rule, we need to reduce its argument with the App rule. We obtain 4 with a trace $\langle 4\rangle$. We return the result of the right hand side of the left and the concatenation of both traces by usage of the LetIn rule, which gives us 4 with a trace $\langle 3; 4\rangle$.

A.3.2 Modules. We now present an example of reduction involving modules. Our example program P is presented in Example 23a. It consists of two declarations: a module declaration X which contains a single declaration a, and the return value of the program, which is equal to $X.a$. It is fairly easy to see that the program P return a value of type `int`, hence we focus on the execution of P, which is presented in Example 23b. The derivation is slightly simplified for clarity. In particular, rules such as EmptyStruct are elided. The first step is to apply the Program and ModuleDecl rules in order to execute the content of each declaration. The declaration of X, on the left side, can be reduced by first applying the Struct rule in order to extract the content of the module structure, then ValDecl, to reduce the declaration of a. These reductions give us the structure value $\{a \mapsto 3\}$. We now execute the declaration of `return`. According to the ModuleDecl rule, we must do so in a new environment containing X: $\{X \mapsto \{a \mapsto 3\}\}$. In order to reduce $X.a$, we must use the QualModVar rule, which reduces qualified variables. This means we first reduce X, which according to the environment gives us $\{a \mapsto 3\}$, noted V. We then look up a in V, which returns 3. To return, we first compose the resulting structure value from both declaration: $\{X \mapsto \{a \mapsto 3\}\} + \{\texttt{return} \mapsto 3\}$. We then lookup `return` in this structure, which gives us 3.

$$\textsc{LetIn}\ \dfrac{\textsc{Const}\ \dfrac{\textit{TypeOf}(\texttt{print}) > \texttt{int} \to \texttt{int}}{\triangleright\, \texttt{print} : \texttt{int} \to \texttt{int}} \quad \textsc{Const}\ \dfrac{\textit{TypeOf}(3) = \texttt{int}}{\triangleright\, 3 : \texttt{int}}}{\textsc{App}\ \dfrac{\triangleright\, (\texttt{print}\ 3) : \texttt{int}}{\triangleright\, \texttt{let}\ x = (\texttt{print}\ 3)\ \texttt{in}\ (\texttt{print}\ (x+1)) : \texttt{int}}} \quad \textsc{App}\ \dfrac{\vdots \quad \vdots}{(\texttt{val}\ x : \texttt{int}) \triangleright (\texttt{print}\ (x+1)) : \texttt{int}}$$

Example 21. Typing derivation for $e - \triangleright\, e : \texttt{int}$

$$\textsc{Let}\ \dfrac{\textsc{Delta}\ \dfrac{\delta(\texttt{print}, 3) = 3, \langle 3 \rangle}{(\texttt{print}\ 3) \Rightarrow 3, \langle 3 \rangle} \quad \textsc{App}\ \dfrac{\textsc{Delta}\ \dfrac{\delta(\texttt{print}, 4) = 4, \langle 4 \rangle}{(\texttt{print}\ 4) \Rightarrow 4, \langle 4 \rangle} \quad \dfrac{\vdots}{x + 1 \xrightarrow{\{x \mapsto 3\}} 4, \langle\rangle}}{(\texttt{print}\ (x+1)) \xrightarrow{\{x \mapsto 3\}} 4, \langle 4 \rangle}}{\texttt{let}\ x = (\texttt{print}\ 3)\ \texttt{in}\ (\texttt{print}\ (x+1)) \Rightarrow 4, \langle 3; 4 \rangle}$$

Example 22. Execution derivation for $e - e \Rightarrow 4, \langle 3; 4 \rangle$

```
prog
 module X = struct let a = 3 end
 let return = X.a
end
```

(a) The program P

$$\textsc{Program}\ \dfrac{\textsc{ModuleDecl}\ \dfrac{\textsc{Struct}\ \dfrac{\textsc{ValDecl}\ \dfrac{3 \Rightarrow 3, \langle\rangle}{\texttt{let}\ a = 3 \Rightarrow \{a \mapsto 3\}, \langle\rangle}}{\begin{pmatrix}\texttt{struct} \\ \texttt{let}\ a = 3 \\ \texttt{end}\end{pmatrix} \Rightarrow \{a \mapsto 3\}, \langle\rangle} \quad \textsc{ValDecl}\ \dfrac{\textsc{QualModVar}\ \dfrac{\textsc{ModVar}\ \dfrac{\rho(X) = V}{X \xRightarrow{\rho} V \equiv \{a \mapsto 3\}, \langle\rangle} \quad V(a) = 3}{X.a \xrightarrow{\{X \mapsto \{a \mapsto 3\}\}} 3, \langle\rangle}}{\texttt{let return} = X.a \xrightarrow{\{X \mapsto \{a \mapsto 3\}\}} \{\texttt{return} \mapsto 3\}, \langle\rangle}}{\begin{pmatrix}\texttt{module}\ X = \texttt{struct let}\ a = 3\ \texttt{end} \\ \texttt{let return} = X.a\end{pmatrix} \Rightarrow \{X \mapsto \{a \mapsto 3\}\} + \{\texttt{return} \mapsto 3\}, \langle\rangle}}{P \Rightarrow 3, \langle\rangle}$$

(b) Execution of P

Example 23. Example of execution with modules

A.3.3 Notes on Soundness. Soundness properties, which correspond to the often misquoted "Well typed programs cannot go wrong.", have been proven for many variants of the ML language. Unfortunately, stating and proving the soundness property for big step semantics and ML modules requires a fairly large amount of machinery which we do not attempt to provide here. Instead, we give pointers to various relevant work containing such proofs.

Soundness for a small step semantics of our expression language is provided by Wright and Felleisen [1994]. At a larger scale, Owens [2008] proves the soundness of a small step semantics for a very large portion of the OCaml expression language using the Locally Nameless Coq framework

[Aydemir et al. 2008]. Soundness of a big step semantics has been proved and mechanized for several richer languages [Amin and Rompf 2017; Garrigue 2009; Lee et al. 2007; Owens et al. 2016; Tofte 1988].

Unfortunately, as far as we are aware, soundness of Leroy's module language with higher order applicative functors has not be proved directly and is a fairly delicate subject. The most recent work of interest is [Rossberg et al. 2014], which presents an elaboration scheme from ML modules, including applicative OCaml-style modules, into System F_ω. Soundness then relies on soundness of the elaboration (provided in the article) and soundness of System F_ω. In this work, the applicative/generative behavior of functors is decided depending on its purity, which is much more precise than what is done in OCaml.

$$\textsc{ModVar}\ \frac{(\texttt{module}\ X_i : \mathcal{M}) \in \Gamma}{\Gamma \blacktriangleright X_i : \mathcal{M}} \qquad \textsc{QualModVar}\ \frac{\Gamma \blacktriangleright p : (\texttt{sig}\ \mathcal{S}_1; \texttt{module}\ X_i : \mathcal{M}; \mathcal{S}_2\ \texttt{end})}{\Gamma \blacktriangleright p.X : \mathcal{M}[n_i \mapsto p.n \mid n_i \in \mathrm{BV}(\mathcal{S}_1)]} \qquad \textsc{Strength}\ \frac{\Gamma \blacktriangleright p : \mathcal{M}}{\Gamma \blacktriangleright p : \mathcal{M}/p}$$

$$\frac{\Gamma \blacktriangleright M : \mathcal{M}' \qquad \Gamma \blacktriangleright \mathcal{M}' <: \mathcal{M}}{\Gamma \blacktriangleright M : \mathcal{M}} \qquad \frac{\Gamma \blacktriangleright M_1 : \texttt{functor}(X_i : \mathcal{M})\mathcal{M}' \qquad \Gamma \blacktriangleright M_2 : \mathcal{M}}{\Gamma \blacktriangleright M_1(M_2) : \mathcal{M}'[X_i \mapsto M_2]}$$

$$\frac{\Gamma \vDash \mathcal{M} \qquad X_i \notin \mathrm{BV}(\Gamma) \qquad \Gamma; (\texttt{module}\ X_i : \mathcal{M}) \blacktriangleright M : \mathcal{M}'}{\Gamma \blacktriangleright \texttt{functor}(X_i : \mathcal{M})M : \texttt{functor}(X_i : \mathcal{M})\mathcal{M}'} \qquad \frac{\Gamma \vDash \mathcal{M} \qquad \Gamma \blacktriangleright M : \mathcal{M}}{\Gamma \blacktriangleright (M : \mathcal{M}) : \mathcal{M}}$$

$$\frac{\Gamma \triangleright e : \tau \qquad x_i \notin \mathrm{BV}(\Gamma) \qquad \Gamma; (\texttt{val}\ x_i : \mathit{Close}(\tau, \Gamma)) \blacktriangleright S : \mathcal{S}}{\Gamma \blacktriangleright (\texttt{let}\ x_i = e; s) : (\texttt{val}\ x_i : \tau; \mathcal{S})}$$

$$\frac{\Gamma \vDash \tau \qquad \mathrm{t_i} \notin \mathrm{BV}(\Gamma) \qquad \Gamma; (\texttt{type}\ \mathrm{t_i} = \tau) \blacktriangleright S : \mathcal{S}}{\Gamma \blacktriangleright (\texttt{type}\ \mathrm{t_i} = \tau; s) : (\texttt{type}\ \mathrm{t_i} = \tau; \mathcal{S})}$$

$$\frac{\Gamma \blacktriangleright M : \mathcal{M} \qquad X_i \notin \mathrm{BV}(\Gamma) \qquad \Gamma; (\texttt{module}\ X_i : \mathcal{M}) \blacktriangleright S : \mathcal{S}}{\Gamma \blacktriangleright (\texttt{module}\ X_i = M; s) : (\texttt{module}\ X_i : \mathcal{M}; \mathcal{S})} \qquad \frac{\Gamma \blacktriangleright S : \mathcal{S}}{\Gamma \blacktriangleright \texttt{struct}\ S\ \texttt{end} : \texttt{sig}\ \mathcal{S}\ \texttt{end}}$$

$$\frac{}{\Gamma \blacktriangleright \varepsilon : \varepsilon}$$

Fig. 46. Module typing rules – $\Gamma \blacktriangleright m : \mathcal{M}$

$$\textsc{SubStruct}\ \frac{\pi : [1; m] \to [1; n] \qquad \forall i \in [1; m],\ \Gamma; \mathcal{D}_1; \ldots; \mathcal{D}_n \blacktriangleright \mathcal{D}_{\pi(i)} <: \mathcal{D}'_i}{\Gamma \blacktriangleright (\texttt{sig}\ \mathcal{D}_1; \ldots; \mathcal{D}_n\ \texttt{end}) <: (\texttt{sig}\ \mathcal{D}'_1; \ldots; \mathcal{D}'_m\ \texttt{end})}$$

$$\frac{\Gamma \triangleright \tau_1 \approx \tau_2}{\Gamma \blacktriangleright (\texttt{val}\ x_i : \tau_1) <: (\texttt{val}\ x_i : \tau_2)} \qquad \frac{\Gamma \blacktriangleright \mathcal{M}_1 <: \mathcal{M}_2}{\Gamma \blacktriangleright (\texttt{module}\ X_i : \mathcal{M}_1) <: (\texttt{module}\ X_i = \mathcal{M}_2)}$$

$$\frac{\Gamma \blacktriangleright \mathcal{M}'_a <: \mathcal{M}_a \qquad \Gamma, (\texttt{module}\ X : \mathcal{M}'_a) \blacktriangleright \mathcal{M}_r <: \mathcal{M}'_r}{\Gamma \blacktriangleright \texttt{functor}(X : \mathcal{M}_a)\mathcal{M}_r <: \texttt{functor}(X : \mathcal{M}'_a)\mathcal{M}'_r} \qquad \frac{\Gamma \triangleright \tau_1 \approx \tau_2}{\Gamma \blacktriangleright (\texttt{type}\ \mathrm{t_i} = \tau_1) <: (\texttt{type}\ \mathrm{t_i} = \tau_2)}$$

$$\frac{}{\Gamma \blacktriangleright (\texttt{type}\ \mathrm{t_i}) <: (\texttt{type}\ \mathrm{t_i})} \qquad \frac{\Gamma \triangleright \mathrm{t_i} \approx \tau}{\Gamma \blacktriangleright (\texttt{type}\ \mathrm{t_i}) <: (\texttt{type}\ \mathrm{t_i} = \tau)} \qquad \frac{}{\Gamma \blacktriangleright (\texttt{type}\ \mathrm{t_i} = \tau_1) <: (\texttt{type}\ \mathrm{t_i})}$$

Fig. 47. Module subtyping rules – $\Gamma \blacktriangleright \mathcal{M} <: \mathcal{M}'$

$$\frac{\Gamma \vDash \mathcal{S}}{\Gamma \vDash \texttt{sig}\ \mathcal{S}\ \texttt{end}} \qquad \frac{}{\Gamma \vDash \varepsilon} \qquad \frac{\Gamma \vDash \mathcal{M}_a \quad x_i \notin \mathrm{BV}(\Gamma) \quad \Gamma; (\texttt{module}\ X_i : \mathcal{M}_a) \vDash \mathcal{M}_r}{\Gamma \vDash \texttt{functor}(X_i : \mathcal{M}_a)\mathcal{M}_r}$$

$$\frac{\mathrm{t_i} \notin \mathrm{BV}(\Gamma) \quad \Gamma; (\texttt{type}\ \mathrm{t_i}) \vDash \mathcal{S}}{\Gamma \vDash \texttt{type}\ \mathrm{t_i};\ \mathcal{S}} \qquad \frac{\Gamma \vDash \mathcal{M} \quad x_i \notin \mathrm{BV}(\Gamma) \quad \Gamma; (\texttt{module}\ X_i : \mathcal{M}) \vDash \mathcal{S}}{\Gamma \vDash \texttt{module}\ X_i : \mathcal{M};\ \mathcal{S}}$$

$$\frac{\Gamma \vDash \tau \quad \mathrm{t_i} \notin \mathrm{BV}(\Gamma) \quad \Gamma; (\texttt{type}\ \mathrm{t_i} = \tau) \vDash \mathcal{S}}{\Gamma \vDash \texttt{type}\ \mathrm{t_i} = \tau;\ \mathcal{S}} \qquad \frac{\Gamma \vDash \tau \quad x_i \notin \mathrm{BV}(\Gamma) \quad \Gamma; (\texttt{val}\ x_i : \tau) \vDash \mathcal{S}}{\Gamma \vDash \texttt{val}\ x_i : \tau;\ \mathcal{S}}$$

Fig. 48. Module type validity rules – $\Gamma \vDash \mathcal{M}$

QualVar
$$\frac{\Gamma \blacktriangleright p : (\texttt{sig}\ \mathcal{S}_1; \texttt{val}\ x_i : \tau; \mathcal{S}_2\ \texttt{end})}{\Gamma \triangleright p.x : \tau[\mathrm{n_i} \mapsto \mathrm{p.n} \mid \mathrm{n_i} \in \mathrm{BV}(\mathcal{S}_1)]}$$

QualDefTypeEq
$$\frac{\Gamma \blacktriangleright p : (\texttt{sig}\ \mathcal{S}_1; \texttt{type}\ \mathrm{t_i} = \tau; \mathcal{S}_2\ \texttt{end})}{\Gamma \triangleright (\tau_i)\mathrm{p.t} \approx \tau[n_i \mapsto p.n \mid n_i \in \mathrm{BV}(\mathcal{S}_1)][\alpha_i \mapsto \tau_i]_i}$$

QualAbsTypeEq
$$\frac{\Gamma \blacktriangleright p : (\texttt{sig}\ \mathcal{S}_1; \texttt{type}\ \mathrm{t_i}; \mathcal{S}_2\ \texttt{end}) \qquad \forall i,\ \Gamma \triangleright \tau_i \approx \tau_i'}{\Gamma \triangleright (\tau_i)\mathrm{p.t} \approx (\tau_i')\mathrm{p.t}}$$

Fig. 49. Additional typing rules for the expression language

$$\begin{aligned}
\varepsilon/p &= \varepsilon \\
(\texttt{sig}\ \mathcal{S}\ \texttt{end})/p &= \texttt{sig}\ \mathcal{S}/p\ \texttt{end} \\
(\texttt{module}\ X_i = \mathcal{M}; \mathcal{S})/p &= \texttt{module}\ X_i = \mathcal{M}/p; \mathcal{S}/p \\
(\texttt{type}\ \mathrm{t_i} = \tau; \mathcal{S})/p &= \texttt{type}\ \mathrm{t_i} = (\alpha^*)\mathrm{p.t}; \mathcal{S}/p \\
(\texttt{type}\ \mathrm{t_i}; \mathcal{S})/p &= \texttt{type}\ \mathrm{t_i} = (\alpha^*)\mathrm{p.t}; \mathcal{S}/p \\
(\texttt{val}\ x_i : \tau; \mathcal{S})/p &= \texttt{val}\ x_i : \tau; \mathcal{S}/p \\
(\texttt{functor}(X_i : \mathcal{M})\mathcal{M}')/p &= \texttt{functor}(X_i : \mathcal{M})(\mathcal{M}'/p(X_i))
\end{aligned}$$

Fig. 50. Module strengthening operation – $\mathcal{M}/p$

PrintTy
$$\mathit{TypeOf}(\texttt{print}) = \forall \alpha.(\alpha \to \alpha)$$

PrintExec
$$\delta(\texttt{print}, v) = v, \langle v \rangle$$

Fig. 51. Typing and execution rules for print

Expressions

$$v ::= c \quad \text{(Constant)} \qquad | \ \lambda x.\rho.e \quad \text{(Function)}$$

Modules

$$V ::= \{\ V_b^*\ \} \quad \text{(Structure)} \qquad | \ \texttt{functor}(\rho)(X_i : \mathcal{M})M$$

Bindings

$$V_b ::= \{x_i \mapsto v\} \quad \text{(Values)} \qquad | \ \{X_i \mapsto V\} \quad \text{(Modules)}$$

Fig. 52. ML values

$$\textsc{Var}\ \frac{\rho(x) = v}{x \overset{\rho}{\Longrightarrow} v, \langle\rangle} \qquad \textsc{QualVar}\ \frac{p \overset{\rho}{\Longrightarrow} V, \theta \qquad V(x) = v}{p.x \overset{\rho}{\Longrightarrow} v, \theta} \qquad \textsc{Constant}\ \frac{}{c \overset{\rho}{\Longrightarrow} c, \langle\rangle} \qquad \textsc{Closure}\ \frac{}{\lambda x.e \overset{\rho}{\Longrightarrow} \lambda x.\rho.e, \langle\rangle}$$

$$\textsc{LetIn}\ \frac{e' \overset{\rho}{\Longrightarrow} v', \theta \qquad e \overset{\rho + \{x \mapsto v'\}}{\Longrightarrow} v, \theta'}{(\texttt{let}\ x = e'\ \texttt{in}\ e) \overset{\rho}{\Longrightarrow} v, \theta \mathbin{@} \theta'} \qquad \textsc{App}\ \frac{e \overset{\rho}{\Longrightarrow} v, \theta \qquad e' \overset{\rho}{\Longrightarrow} v', \theta' \qquad (v\ v') \overset{\rho}{\Longrightarrow} v'', \theta''}{(e\ e') \overset{\rho}{\Longrightarrow} v'', \theta \mathbin{@} \theta' \mathbin{@} \theta''}$$

$$\textsc{Beta}\ \frac{e \overset{\rho' + \{x \mapsto v\}}{\Longrightarrow} v', \theta}{(\lambda x.\rho'.e\ v) \overset{\rho}{\Longrightarrow} v', \theta} \qquad \textsc{Y}\ \frac{(v\ \lambda x.(\texttt{Y}\ v\ x)) \overset{\rho}{\Longrightarrow} v', \theta}{(\texttt{Y}\ v) \overset{\rho}{\Longrightarrow} v', \theta} \qquad \textsc{Delta}\ \frac{\delta(c, v) = v', \theta}{(c\ v) \overset{\rho}{\Longrightarrow} v', \theta}$$

$$\textsc{ModVar}\ \frac{\rho(X) = V}{X \overset{\rho}{\Longrightarrow} V, \langle\rangle} \qquad \textsc{QualModVar}\ \frac{p \overset{\rho}{\Longrightarrow} V', \theta \qquad V'(X) = V}{p.X \overset{\rho}{\Longrightarrow} V, \theta} \qquad \textsc{Struct}\ \frac{S \overset{\rho}{\Longrightarrow} V_s, \theta}{(\texttt{struct}\ S\ \texttt{end}) \overset{\rho}{\Longrightarrow} V_s, \theta} \qquad \textsc{EmptyStruct}\ \frac{}{\varepsilon \overset{\rho}{\Longrightarrow} \{\}, \langle\rangle}$$

$$\textsc{ModClosure}\ \frac{}{\texttt{functor}(X : \mathcal{M})M \overset{\rho}{\Longrightarrow} \texttt{functor}(\rho)(X : \mathcal{M})M, \langle\rangle} \qquad \textsc{ModConstr}\ \frac{M \overset{\rho}{\Longrightarrow} V, \theta}{(M : \mathcal{M}) \overset{\rho}{\Longrightarrow} V, \theta}$$

$$\textsc{ModBeta}\ \frac{M \overset{\rho}{\Longrightarrow} \texttt{functor}(\rho')(X : \mathcal{M})M_f, \theta \qquad M' \overset{\rho}{\Longrightarrow} V', \theta' \qquad M_f \overset{\rho' + \{X \mapsto V'\}}{\Longrightarrow} V'', \theta''}{M(M') \overset{\rho}{\Longrightarrow} V'', \theta \mathbin{@} \theta' \mathbin{@} \theta''}$$

$$\textsc{TypeDecl}\ \frac{S \overset{\rho}{\Longrightarrow} V_s, \theta}{(\texttt{type}\ \mathrm{t_i} = \tau; S) \overset{\rho}{\Longrightarrow} V_s, \theta} \qquad \textsc{ModuleDecl}\ \frac{M \overset{\rho}{\Longrightarrow} V, \theta \qquad S \overset{\rho + \{X_i \mapsto V\}}{\Longrightarrow} V_s, \theta'}{(\texttt{module}\ X_i = M; S) \overset{\rho}{\Longrightarrow} \{X \mapsto V\} + V_s, \theta \mathbin{@} \theta'}$$

$$\textsc{ValDecl}\ \frac{e \overset{\rho}{\Longrightarrow} v, \theta \qquad S \overset{\rho + \{x_i \mapsto v\}}{\Longrightarrow} V_s, \theta'}{(\texttt{let}\ x_i = e; S) \overset{\rho}{\Longrightarrow} \{x \mapsto v\} + V_s, \theta'} \qquad \textsc{Program}\ \frac{S \overset{\rho}{\Longrightarrow} V_s, \theta}{\texttt{prog}\ S\ \texttt{end} \overset{\rho}{\Longrightarrow} V_s(\texttt{return}), \theta}$$

Fig. 53. Big step semantics – $e \overset{\rho}{\Longrightarrow} v, \theta$

B SEMANTICS PRESERVATION

We now prove Theorem 4 which states that given an Eliom$_\varepsilon$ program P, the trace of its execution is the same as the concatenation of the traces of $\langle P \rangle_s$ and $\langle P \rangle_c$. More formally:

Theorem 6 (Compilation preserves semantics)**.** Given sets of constants where converters are well-behaved, given an Eliom$_\varepsilon$ program P respecting the slicability hypothesis and such that $P \overset{\{\}}{\Longrightarrow} v, \theta$ then

$$\langle P \rangle_s \overset{\{\}}{\Longrightarrow}_{\mathrm{ML_s}} (), \xi, \zeta, \theta_s \qquad \langle P \rangle_c, \xi \overset{\{\} \mid \zeta \to \gamma}{\Longrightarrow}_{\mathrm{ML_c}} v, \xi', \theta_c \qquad \theta = \theta_s \mathbin{@} \theta_c$$

B.1 Hoisting

In Section 5.2, we mentioned that a useful property of injections and fragments is that they can be partially lifted outside sections. This property can be used to simplify the simulation proofs. We consider the code transformation that hoists the content of injections out of fragments, client declarations and mixed functors in a way that preserve semantics. This transformation can be decomposed in two parts.

Injections. We decompose injections inside fragments and client declarations into simpler components. For example, the Eliom$_\varepsilon$ piece of code presented in Example 24a is decomposed in Example 24b by moving out the application of the converter and leaving only a call to the `serial` converter. All injections using a converter than is not `serial` nor `frag` can be decomposed in such a way.

Since injections can only be used on variables or constants and that no server bindings can be introduced inside a fragment, scoping is preserved. Furthermore, by definition of converters and their client and server components, this transformation preserves typing. It also preserves the dynamic semantics as long as the order of hoisting correspond to the order of evaluation. This can be seen by inspecting the reduction relation for server code under client contexts $\Rightarrow_{c/s}$. Finally, it trivially preserves the semantics of the compiled program since it corresponds exactly to how converters are decomposed during compilation.

```
let a = 1 + 2 in
{{ 3 + int%a }}
```

(a) Fragment with injections

```
let a = 1 + 2 in
let a' = (int^s a) in
{{ 3 + (int^c serial%a') }}
```

(b) Fragment with hoisted injections

Example 24. Hoisting on fragments

This allows us to assume that reduction of server code in client context only uses variable lookup and never leads to any evaluation. In particular, this will avoid having to deal with the case of fragments being executed inside the reduction of another fragment (to see why this could happen, consider the case of a converter of type $\forall \alpha_c.(\mathrm{unit} \to \{\alpha_c\}) \rightsquigarrow \alpha_c$).

In the rest of this section, we assume that reductions of the Eliom$_\varepsilon$ rule Fragment are always of the following shape:

$$\frac{e \overset{\rho_c}{\Longrightarrow}_{c/s} \overline{e}, \varepsilon, \langle\rangle}{\{\{\ e\ \}\} \overset{\rho_s}{\Longrightarrow}_s \mathbf{r}, (\mathtt{bind}\ \mathbf{r} = \overline{e}), \langle\rangle} \text{ where } \overline{e} = e[f_i \% x_i \mapsto \rho_s(x_i)]_i \text{ and } f_i \in \{\mathtt{serial}, \mathtt{frag}\}$$

and that reductions of the Eliom$_\varepsilon$ rule ClientDecl are always of the following shape:

$$\frac{}{D_c \overset{\rho_s}{\Longrightarrow}_{c/s} \overline{D_c}, \varepsilon, \langle\rangle} \text{ where } \overline{D_c} = D_c[f_i\%x_i \mapsto \rho_s(x_i)]_i \text{ and } f_i \in \{\texttt{serial}, \texttt{frag}\}$$

Injections inside mixed modules. We also hoist injections completely out of mixed contexts to the outer englobing scope. For example in the functor presented in Example 25a, we can lift the injection out of the functor, as show in Example 25b. This is valid since injections can only reference content outside of the functor, by typing. Semantics is similarly preserved since injections inside functors are reduced immediately when encountering a functor, as per rule ModClosure in Figure 32.

This allows us to assume that the reduction of a mixed module will never lead to the reduction of an injection.

```
let_s x = ...
module_m F(X : M) = struct
  let_c y = f%x
end
```

(a) Mixed functor with injections

```
let_s x = ...
let_c y' = f%x
module_m F(X : M) = struct
  let_c y = y'
end
```

(b) Mixed functor with hoisted injections

Example 25. Hoisting on mixed modules

B.2 Preliminaries

Let us start with some naming conventions. Identifiers with a hat, such as $\widehat{\gamma}$, are related to the compiled semantics. For example, while the server environment for the interpreted semantics is noted ρ_s, the environment for the execution of the target language ML$_s$ is noted $\widehat{\rho_s}$. This naming convention is only for ease of reading and does not apply a formal relation between the objects with and without hats, unless indicated explicitly.

B.2.1 Remarks about global environments. Let us make some preliminary remarks about global environments in the Eliom$_\varepsilon$ client generated programs and in ML$_c$.

Given a global environment γ resulting of $\Rightarrow_c$, it contains only two kinds of references:

- Closure fragments, noted **f**, which come from the execution of `bind env`. The associated value is always a environment (*i.e.*, a signature).
- Fragment values, noted **r**, which come from the execution of `bind with`.

In the rest of this section, we consider that we can always decompose global environments γ in two parts: a fragment value environment γ_r containing all the references **r** that were produced by `bind with` and a fragment closure environment γ_f containing only binding of the form $\{\mathbf{f} \mapsto \rho\}$ that were produced by `bind env`.

Similarly, given a global environment $\widehat{\gamma}$ used in ML$_c$. There are only three kind of references:

- Closure fragments, noted **f**, which come from the slicing of syntactic fragments in the source program. The associated value is always a closure.
- Fragment values, noted **r** come from the execution of fragments in the fragment queue.

- Injections, noted **x**. The associated values must be serializable, and hence can only be references or constants in $Const_b$.

In the rest of this section, we consider that we can always decompose global compiled environment $\widehat{\gamma}$ into a fragment closure environment $\widehat{\gamma}_f$, a fragment value environment $\widehat{\gamma}_r$ and an injection environment ζ.

B.2.2 Client equivalence.

Definition 2 (Client values equivalence)**.** Given v an Eliom client value, v' an $\mathrm{ML_c}$ value and ζ an environment of references, v and v' are equivalent under ζ, noted $v \simeq^c_\zeta v'$, if and only if they are equals after substitution by ζ: $v[\zeta] = v'[\zeta]$.

We extend this notation to environments and traces.

Definition 3 (Global environment equivalence)**.** We say that an Eliom_ε global environment $\gamma = \gamma_f \cup \gamma_r$ and an $\mathrm{ML_c}$ global environment $\widehat{\gamma} = \widehat{\gamma}_f \cup \widehat{\gamma}_r \cup \zeta$ are synchronized if and only if the following conditions hold.

- The reference environments are equivalent: $\gamma_r \simeq^c_\zeta \widehat{\gamma}_r$
- The domains of γ_f and $\widehat{\gamma}_f$ coincides, and:
  - For each $\mathbf{f}$ in these environments such that $\gamma_f(\mathbf{f}) = \rho$ and that $\widehat{\gamma}_f(\mathbf{f}) = \lambda x_0 \dots x_n.\widehat{\rho}.e$, then the following property must hold.
    We must have that $\rho \simeq^c_\zeta \widehat{\rho}$ and that for all $v_0, \dots, v_n$, $\widehat{v}_0, \dots, \widehat{v}_n$ such that for all i, $v_i \simeq^c_\zeta \widehat{v}_i$; then:
    $$e[x_i \mapsto v_i]_i \xrightarrow{\rho \mid \gamma \to \gamma}_c v, \theta \implies (\lambda \overline{x}_i.\widehat{\rho}.e\ \widehat{v}_0 \dots \widehat{v}_n) \xrightarrow{\mid \widehat{\gamma} \to \widehat{\gamma}}_{\mathrm{ML_c}} \widehat{v}, \widehat{\theta}$$
    with $v \simeq^c_\zeta \widehat{v}$ and $\theta \simeq^c_\zeta \widehat{\theta}$
  - For each $\mathbf{F}$ in these environments such that $\gamma_f(\mathbf{F}) = \rho$ and that $\widehat{\gamma}_f(\mathbf{F}) = \texttt{functor}(\widehat{\rho})(Y_i : \mathcal{M}_i)_i M$, we have $\rho \simeq^c_\zeta \widehat{\rho}$.

Definition 4 (Fragment closure environment)**.** We consider that $\widehat{\gamma}_f$ is a fragment closure environment for the Eliom_ε server expression e_s, noted $FCE(\widehat{\gamma}_f, e_s)$, if for each $\{\mathbf{f} \mapsto \lambda \overline{x}_i.\widehat{\rho}.e'\}$ in $\widehat{\gamma}_f$, for each $\{\{\ e\ \}\}_\mathbf{f}$ in $\mathrm{FRAGS}(e_s)$ we have $e' = e[f_i \% x_i \mapsto x_i]_i$.

Definition 5 (Functor closure environment)**.** We consider that $\widehat{\gamma}_f$ is a functor closure environment for the Eliom_ε module expression M, noted $FCE(\widehat{\gamma}_f, M)$, if for each $\left\{\mathbf{F} \mapsto \texttt{functor}(\widehat{\rho})(Y_i : \mathcal{M}_i)_i \widehat{S}\right\}$ in $\widehat{\gamma}_f$, for each $(\texttt{struct}\ S\ \texttt{end}_\mathbf{F})$ in M_s we have $\widehat{S} = \langle S \rangle_c$. Additionally, we require that $\widehat{\gamma}_f$ be a fragment closure environment for each expression contained in S.

In the rest of this section, we use the same notation for both properties. We extend this notation to server declarations, server values (by looking under closures) and server environments.

Lemma 1 (Reduction up to equivalence)**.** Given $\rho, \widehat{\rho}, \gamma = \gamma_f \cup \gamma_r, \widehat{\gamma} = \widehat{\gamma}_f \cup \widehat{\gamma}_r \cup \zeta$, e and $\widehat{e}$ such that:

$$\rho \simeq^c_\zeta \widehat{\rho} \qquad \gamma \simeq^c_\zeta \widehat{\gamma} \qquad e[\zeta] = \widehat{e}[\zeta] \qquad e \xrightarrow{\rho \mid \gamma \to \gamma}_c v, \theta$$

Then we have:

$$\widehat{e} \xrightarrow{\widehat{\rho} \mid \widehat{\gamma} \to \widehat{\gamma}}_{\mathrm{ML_c}} \widehat{v}, \widehat{\theta} \qquad v \simeq^c_\zeta \widehat{v} \qquad \theta \simeq^c_\zeta \widehat{\theta}$$

Proof. The only difference between Eliom$_\varepsilon$ client expressions and ML$_c$ expressions are the presence of extra references for injections in ML$_c$. Indeed, syntactic injections have been removed either by the server execution or by compilation and `bind` constructs are only accessible at the module level. Since we assume that the original expression e and the compiled expression $\widehat{e}$ are the same up to the injection environment ζ, we can trivially mimic the execution of e in $\widehat{e}$ by induction. □

B.2.3 Server equivalence.

Definition 6 (Server value equivalence)**.** Given v an Eliom server value, $\widehat{v}$ an ML$_s$ value. We say they are equivalent, noted $v \simeq^s \widehat{v}$ if and only if

$$v\left[\{\{\ e_i\ \}\}_{\mathbf{f}} \mapsto \texttt{fragment}\ \mathbf{f}\ \overline{x_{i,j}}\right]_i = \widehat{v} \qquad \text{where } \overline{x_{i,j}} = \text{INJS}(e_i)$$

We extend this notation to environments and traces.

B.3 Server expressions and structures

We first look at server expressions and structures. By definition of the server reduction relation for Eliom$_\varepsilon$, the emitted program is a series of binds.

Lemma 2 (Server expressions are simulable)**.** We consider an Eliom$_\varepsilon$ server expression e; the Eliom$_\varepsilon$ environments ρ_s, ρ_c and $\gamma = \gamma_f \cup \gamma_r$; the target environment $\widehat{\rho}_s$, $\widehat{\rho}_c$ and $\widehat{\gamma} = \widehat{\gamma}_f \cup \widehat{\gamma}_r \cup \zeta$.

If the expression e has valid server and client executions:

$$e \xRightarrow{\rho_s}_s v, \mu, \theta_s \qquad\qquad \mu \xRightarrow{\rho_c \mid \gamma \to \gamma'}_c \{\}, \theta_c$$

and the following invariants hold:

$$\widehat{\rho}_c \simeq^c_\zeta \rho_c \qquad \widehat{\rho}_s \simeq^s \rho_s \qquad \widehat{\gamma} \simeq^c \gamma \qquad FCE(\widehat{\gamma}_f, e) \qquad FCE(\widehat{\gamma}_f, \rho_s)$$

Then $\widehat{e} = e\left[\{\{\ e_i\ \}\}_{\mathbf{f}} \mapsto \texttt{fragment}\ \mathbf{f}\ \overline{x_{i,j}}\right]_i$ has an equivalent execution.

$$\frac{}{\widehat{e} \xRightarrow{\widehat{\rho}_s}_{\text{ML}_s} \widehat{v}, \xi_\bullet, \{\}, \widehat{\theta}_s} \qquad\qquad \frac{}{\texttt{exec}\ (), \xi_\bullet +\!\!+ \texttt{end} \xRightarrow{\widehat{\rho}_c \mid \widehat{\gamma} \to \widehat{\gamma}'}_{\text{ML}_c} \varepsilon, [\ ], \widehat{\theta}_c}$$

with the following invariants:

$$\widehat{\gamma}' \simeq^c \gamma' \qquad FCE(\widehat{\gamma}'_f, v) \qquad \widehat{v} \simeq^s v \qquad \widehat{\theta}_s \simeq^s \theta_s \qquad \widehat{\theta}_c \simeq^c_\zeta \theta_c$$

Proof. We consider an expression e; the Eliom$_\varepsilon$ environments ρ_s, ρ_c and $\gamma = \gamma_f \cup \gamma_r$; the target environment $\widehat{\rho}_s$, $\widehat{\rho}_c$ and $\widehat{\gamma} = \widehat{\gamma}_f \cup \widehat{\gamma}_r \cup \zeta$. such that

$$\widehat{\gamma} \simeq^c \gamma \qquad \widehat{\rho}_c \simeq^c_\zeta \rho_c \qquad \widehat{\rho}_s \simeq^s \rho_s \qquad FCE(\widehat{\gamma}_f, e) \qquad FCE(\widehat{\gamma}_f, \rho_s)$$

We will proceed by induction over the executions of e and μ. The only case of interest is when the server expression is a fragment.

• Case $\{\{\ e\ \}\}_{\mathbf{f}}$.

We assume that the following executions hold:

$$\frac{\rho_s(x_i) = v_i}{\{\{\ e\ \}\}_{\mathbf{f}} \xRightarrow{\rho_s}_m \mathbf{r}, \texttt{bind}\ \mathbf{r} = \overline{e}\ \texttt{with}\ \mathbf{f}, \langle\rangle} \qquad \frac{\gamma(\mathbf{f}) = \rho \qquad \overline{e} \xRightarrow{\rho \mid \gamma \to \gamma}_c v_c, \theta_c}{(\texttt{bind}\ \mathbf{r} = \overline{e}\ \texttt{with}\ \mathbf{f}) \xRightarrow{\rho_c \mid \gamma \to \gamma'}_c \{\}, \theta_c}$$

where $\overline{e} = e[f_i \% x_i \mapsto \downarrow v_i]_i$ and $\gamma' = \gamma \cup \{\mathbf{r} \mapsto v_c\}$. We have $\widehat{e}$ equal to $\texttt{fragment}\ \mathbf{f}\ x_1 \ldots x_n$.

We first consider the execution of $\widehat{e}$. We can easily construct the following execution.

$$\frac{\widehat{\rho}_s(x_i) = \widehat{v}_i}{\texttt{fragment f } x_1 \dots x_n \xRightarrow{\widehat{\rho}_s}_{\text{ML}_s} \mathbf{r}, \{\mathbf{r}\mapsto \mathbf{f}\ \downarrow\widehat{v}_1 \dots \downarrow\widehat{v}_n\}, \{\}, \langle\rangle}$$

By hypothesis, for each i, $v_i \simeq^s_{\widehat{\gamma}} \widehat{v}_i$. This gives us that $\downarrow v_i \simeq^c_{\widehat{\gamma}} \downarrow\widehat{v}_i$. We trivially have that $\mathbf{r} \simeq^s_{\widehat{\gamma}} \mathbf{r}$

Let us now look at the client execution. By client execution of μ, $\gamma(\mathbf{f}) = \rho$. Since $\gamma \simeq^c_\zeta \widehat{\gamma}$, we have $\{\mathbf{f}\mapsto\lambda x_0 \dots x_n.\widehat{\rho}.e'\} \in \widehat{\gamma}$ and $\rho \simeq^c_{\widehat{\gamma}} \widehat{\rho}$. Furthermore, since $FCE(\widehat{\gamma}_f, \{\{\ e\ \}\}_\mathbf{f})$, we know that that $e' = e[f_i\%x_i \mapsto x_i]_i$. We have by hypothesis that $\overline{e} \xRightarrow{\rho_c \mid \gamma\to\gamma}_c v_c, \theta_c$. Since $\overline{e} = e'[x_i \mapsto \downarrow v_i]$ and since for all i, $\downarrow v_i \simeq^c_{\widehat{\gamma}} \downarrow\widehat{v}_i$, we can use Lemma 1 to build the following reduction:

$$\frac{\dfrac{\dfrac{e' \xRightarrow{\widehat{\rho}_c\cup\{x_i\mapsto\widehat{v}_i\}_i \mid \widehat{\gamma}\to\widehat{\gamma}}_{\text{ML}_c} \widehat{v}, \widehat{\theta}_c}{\lambda x_1 \dots x_n.\widehat{\rho}.e'\ \downarrow\widehat{v}_1 \dots \downarrow\widehat{v}_n \xRightarrow{\widehat{\rho}_c \mid \widehat{\gamma}\to\widehat{\gamma}}_{\text{ML}_c} \widehat{v}, \widehat{\theta}_c}}{\mathbf{f}\ \downarrow\widehat{v}_1 \dots \downarrow\widehat{v}_n \xRightarrow{\widehat{\rho}_c \mid \widehat{\gamma}\to\widehat{\gamma}}_{\text{ML}_c} \widehat{v}, \widehat{\theta}_c}}{\texttt{exec ()}, \{\mathbf{r}\mapsto\mathbf{f}\ \downarrow\widehat{v}_1 \dots \downarrow\widehat{v}_n\} \xRightarrow{\widehat{\rho}_c \mid \widehat{\gamma}\to\widehat{\gamma}'}_{\text{ML}_c} \varepsilon, [\ ], \widehat{\theta}_c}$$

Where $\widehat{\gamma}' = \widehat{\gamma} \cup \{\mathbf{r}\mapsto\widehat{v}\}$. By Lemma 1, we have that $v \simeq^c_{\widehat{\gamma}} \widehat{v}$ and $\theta_c \simeq^c_{\widehat{\gamma}} \widehat{\theta}_c$. The only part that is changed in γ' and $\widehat{\gamma}'$ is the fragment reference environment, hence we easily have that $\widehat{\gamma}' \simeq^c \gamma'$, which concludes. ■

• Other cases.
In other cases, we first note that references manipulated inside server code can only fragment references $\mathbf{r}$. By hypothesis, the same references are considered before and after compilation. Since the fragment closure environment hypothesis ranges over all server expressions, including the one in closures, it is easy to preserve it during execution. The rest is a very simple induction. ■

□

Corollary 1 (Server module declarations are simulable)**.** We consider an Eliom$_\varepsilon$ server declaration D_s; the Eliom$_\varepsilon$ environments ρ_s, ρ_c and $\gamma = \gamma_f \cup \gamma_r$; the target environment $\widehat{\rho}_s$, $\widehat{\rho}_c$ and $\widehat{\gamma} = \widehat{\gamma}_f \cup \widehat{\gamma}_r \cup \zeta$.

If the expression e has valid server and client executions:

$$D \xRightarrow{\rho_s}_s V, \mu, \theta_s \qquad\qquad \mu \xRightarrow{\rho_c \mid \gamma\to\gamma'}_c \{\}, \theta_c$$

and the following invariants hold:

$$\widehat{\rho}_c \simeq^c_\zeta \rho_c \qquad \widehat{\rho}_s \simeq^s \rho_s \qquad \widehat{\gamma} \simeq^c \gamma \qquad FCE(\widehat{\gamma}_f, D) \qquad FCE(\widehat{\gamma}_f, \rho_s)$$

Then $\widehat{D} = D\big[\{\{\ e_i\ \}\}_\mathbf{f} \mapsto \texttt{fragment f } \overline{x_{i,j}}\big]_i$ have an equivalent execution.

$$\frac{}{\widehat{D} \xRightarrow{\widehat{\rho}_s}_{\text{ML}_s} \widehat{V}, \xi_\bullet, \{\}, \widehat{\theta}_s} \qquad\qquad \frac{}{\texttt{exec ()}, \xi_\bullet +\!\!+ \texttt{end} \xRightarrow{\widehat{\rho}_c \mid \widehat{\gamma}\to\widehat{\gamma}'}_{\text{ML}_c} \varepsilon, [\ ], \widehat{\theta}_c}$$

with the following invariants:

$$\widehat{\gamma}' \simeq^c \gamma' \qquad FCE(\widehat{\gamma}'_f, V) \qquad \widehat{V} \simeq^s V \qquad \widehat{\theta}_s \simeq^s \theta_s \qquad \widehat{\theta}_c \simeq^c_{\zeta'} \theta_c$$

B.4 Mixed structures

Lemma 3 (Structures are simulable)**.** We consider a slicable structure S; the Eliom$_\varepsilon$ environments ρ_s, ρ_c and $\gamma = \gamma_f \cup \gamma_r$; the target environment $\widehat{\rho}_s$, $\widehat{\rho}_c$ and $\widehat{\gamma} = \widehat{\gamma}_f \cup \widehat{\gamma}_r \cup \zeta$.

If the structure has valid server and client executions:

$$S \xRightarrow{\rho_s}_m V_s, \mu, \theta_s \qquad\qquad \mu \xRightarrow{\rho_c \mid \gamma \to \gamma'}_c V_c, \theta_c$$

and the following invariants hold:

$$\widehat{\rho}_c \simeq^c_\zeta \rho_c \qquad \widehat{\rho}_s \simeq^s \rho_s \qquad \widehat{\gamma} \simeq^c \gamma \qquad FCE(\widehat{\gamma}_f, S) \qquad FCE(\widehat{\gamma}_f, \rho_s)$$

then for any ξ, the compiled structures have equivalent executions

$$\langle S \rangle_s \xRightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \widehat{V}_s, \xi_\bullet, \zeta_\bullet, \widehat{\theta}_s \qquad\qquad \langle S \rangle_c, \xi_\bullet +\!\!+ \xi \xRightarrow{\widehat{\rho}_c \mid \zeta_\bullet \cup \widehat{\gamma} \to \widehat{\gamma}'}_{\mathrm{ML_c}} \widehat{V}_c, \xi', \widehat{\theta}_c$$

with the following invariants:

$$\widehat{\gamma}' \simeq^c \gamma' \qquad \widehat{V}_s \simeq^s V_s \qquad \widehat{\theta}_s \simeq^s \theta_s$$
$$FCE(\widehat{\gamma}'_f, V_s) \qquad \widehat{V}_c \simeq^c_{\widehat{\gamma}'} V_c \qquad \widehat{\theta}_c \simeq^c_{\widehat{\gamma}'} \theta_c$$

Proof. We consider a slicable structure S; the Eliom$_\varepsilon$ environments ρ_s, ρ_c and $\gamma = \gamma_f \cup \gamma_r$; the target environment $\widehat{\rho}_s$, $\widehat{\rho}_c$ and and $\widehat{\gamma} = \widehat{\gamma}_f \cup \widehat{\gamma}_r \cup \zeta$. such that

$$\widehat{\rho}_c \simeq^c_{\widehat{\gamma}} \rho_c \qquad \widehat{\rho}_s \simeq^s \rho_s \qquad \widehat{\gamma} \simeq^c \gamma \qquad FCE(\widehat{\gamma}_f, S) \qquad FCE(\widehat{\gamma}_f, \rho_s)$$

We will now proceed by induction over the execution of S.

• Case $S = D_b; S'$ – Base declaration.
We assume that the following executions hold:

$$\frac{D_b \xRightarrow{\rho_s}_b V_s, \varepsilon, \theta_s \qquad S' \xRightarrow{\rho_s + V_s}_m V'_s, \mu, \theta'_s}{D_b; S' \xRightarrow{\rho_s}_m V_s + V'_s, (D_b; \mu), \theta_s \mathbin{@} \theta'_s} \qquad \frac{D_b \xRightarrow{\rho_c \mid \gamma \to \gamma}_c V_c, \theta_c \qquad \mu \xRightarrow{\rho_c + V_c \mid \gamma \to \gamma'}_c V'_c, \theta'_c}{D_b; \mu \xRightarrow{\rho_c \mid \gamma \to \gamma'}_c V_c + V'_c, \theta_c \mathbin{@} \theta'_c}$$

Let us consider the executions of D_b. By definition of base, it contains neither injections nor fragments. By Proposition 5, $D_b \xRightarrow{\rho}_b V_s, \varepsilon, \theta_s$ and $D_b \xRightarrow{\rho_c \mid \gamma \to \gamma}_c V_b, \theta_c$ both correspond to $D_b \xRightarrow{\rho_s} V_s, \varepsilon, \theta_s$ and $D_b \xRightarrow{\rho_c} V_c, \varepsilon, \theta_c$ respectively. By definition, base fragments can't be present, hence we also have $FCE(\widehat{\gamma}_f, V_s)$

Additionally, the compilation functions are the identity on base, which mean that $\langle D_b \rangle_s$ and $\langle D_b \rangle_c$ contains only ML constructs. The reduction relation over $\mathrm{ML_s}$ and $\mathrm{ML_c}$ coincide with the ML one on the ML fragment of the language. Hence, for any ξ, we have $\langle D_b \rangle_s \xRightarrow{\rho_s}_{\mathrm{ML_s}} \widehat{V}_s, [\,], \{\}, \widehat{\theta}_s$ and $\langle D_b \rangle_c, \xi \xRightarrow{\rho_c \mid \widehat{\gamma} \to \widehat{\gamma}}_{\mathrm{ML_c}} \widehat{V}_c, \xi, \widehat{\theta}_c$ with

$$\widehat{V}_s \simeq^s V_s \qquad \widehat{V}_c \simeq^c_{\widehat{\gamma}} V_c \qquad \widehat{\theta}_s \simeq^s \theta_s \qquad \widehat{\theta}_c \simeq^c_{\widehat{\gamma}} \theta_c$$

Let us consider the execution of S' and μ. We easily have the following properties:

$$\widehat{\rho}_c + \widehat{V}_c \simeq^c_\zeta \rho_c + V_c \qquad \widehat{\rho}_s + \widehat{V}_s \simeq^s \rho_s + V_s \qquad \widehat{\gamma} \simeq^c \gamma \qquad FCE(\widehat{\gamma}_f, S) \qquad FCE(\widehat{\gamma}_f, \rho_s + V_s)$$

hence, by induction on the execution of S' and μ', we have $\langle S'\rangle_s \xrightarrow{\widehat{\rho}_s+\widehat{V}_s}_{\mathrm{ML_s}} \widehat{V}'_s, \xi_\bullet, \zeta_\bullet, \widehat{\theta}'_s$ and $\langle S'\rangle_c, \xi_\bullet +\!\!+ \xi \xrightarrow{\widehat{\rho}_c+\widehat{V}_c \mid$
for any ξ, with

$$\widehat{\gamma}' \simeq^c \gamma' \qquad \widehat{V}'_s \simeq^s V'_s \qquad \widehat{\theta}_s \simeq^s \theta_s$$
$$FCE(\widehat{\gamma}'_f, V'_s) \qquad \widehat{V}'_c \simeq^c_{\widehat{\gamma}'} V'_c \qquad \widehat{\theta}'_c \simeq^c_{\widehat{\gamma}'} \theta'_c$$

We can then build the following derivations:

$$\frac{\langle D_b\rangle_s \xrightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \widehat{V}_s, [\,], \{\}, \theta_s \qquad \langle S'\rangle_s \xrightarrow{\widehat{\rho}_s+\widehat{V}_s}_{\mathrm{ML_s}} \widehat{V}'_s, \xi_\bullet, \zeta_\bullet, \widehat{\theta}'_s}{\langle D_b; S'\rangle_s \xrightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \widehat{V}_s + \widehat{V}'_s, \xi_\bullet, \zeta_\bullet, \widehat{\theta}_s \mathbin{@} \widehat{\theta}'_s}$$

$$\frac{\langle D_b\rangle_c, \xi_\bullet +\!\!+ \xi \xrightarrow{\widehat{\rho}_c \mid \widehat{\gamma}\cup\zeta_\bullet\to\widehat{\gamma}\cup\zeta_\bullet}_{\mathrm{ML_c}} \widehat{V}_c, \xi_\bullet +\!\!+ \xi, \widehat{\theta}_c \qquad \langle S'\rangle_c, \xi_\bullet +\!\!+ \xi \xrightarrow{\widehat{\rho}_c+\widehat{V}_c \mid \widehat{\gamma}\cup\zeta_\bullet\to\widehat{\gamma}'}_{\mathrm{ML_c}} \widehat{V}'_c, \xi, \widehat{\theta}'_c}{\langle D_b; S\rangle_c, \xi_\bullet +\!\!+ \xi \xrightarrow{\rho_c \mid \widehat{\gamma}\cup\zeta_\bullet\to\widehat{\gamma}'}_{\mathrm{ML_c}} \widehat{V}_c + \widehat{V}'_c, \xi, \widehat{\theta}_c \mathbin{@} \widehat{\theta}'_c}$$

and the following invariants are easily verified:

$$\widehat{\gamma}' \simeq^c \gamma' \qquad \widehat{V}_s + \widehat{V}'_s \simeq^s V_s + V'_s \qquad \widehat{\theta}_s \mathbin{@} \widehat{\theta}'_s \simeq^s \theta_s \mathbin{@} \theta'_s$$
$$FCE(\widehat{\gamma}'_f, V_s + V'_s) \qquad \widehat{V}_c + \widehat{V}'_c \simeq^c_{\widehat{\gamma}'} V_s + V'_c \qquad \widehat{\theta}_c \mathbin{@} \widehat{\theta}'_c \simeq^c_{\widehat{\gamma}'} \theta_c \mathbin{@} \theta'_c \qquad \blacksquare$$

• Case $S = D_s; S'$ – Server declaration.
We assume that the following executions hold:

$$\frac{D_s \xrightarrow{\rho_s}_s V_s, \mu, \theta_s \qquad S' \xrightarrow{\rho_s+V_s}_m V'_s, \mu', \theta'_s}{D_s; S' \xrightarrow{\rho_s}_m V_s + V'_s, (\texttt{bind env } \mathbf{f_i})_i; \mu; \mu', \theta_s \mathbin{@} \theta'_s}$$

$$\frac{(\texttt{bind env } \mathbf{f_i})_i \xrightarrow{\rho_c \mid \gamma\to\gamma'}_c \{\}, \langle\rangle \qquad \mu \xrightarrow{\rho_c \mid \gamma'\to\gamma''}_c V_c, \theta_c \qquad \mu' \xrightarrow{\rho_c \mid \gamma''\to\gamma'''}_c V'_c, \theta'_c}{(\texttt{bind env } \mathbf{f_i})_i; \mu; \mu' \xrightarrow{\rho_c \mid \gamma\to\gamma''}_c V_c + V'_c, \theta_c \mathbin{@} \theta'_c}$$

Let us note $\{\{\, e_i \,\}\}_{\mathbf{f_i}}$ the fragments syntactically present in D_s. let us note $\{\{\, e_j \,\}\}_{\mathbf{f_j}}$ the fragments *executed* during the reduction of D_s and $\mathbf{r_j}$ the associated fresh variables.

We have the following compilations:

$$\langle D_s; S'\rangle_s = \langle D_s\rangle_s \left[\{\{\, e_i \,\}\}_{\mathbf{f_i}} \mapsto \texttt{fragment } \mathbf{f_i}\ \overline{x_{i,k}}\right]_i ; \texttt{end}\ (); \langle S'\rangle_s$$
$$\langle D_s; S'\rangle_c = \left(\texttt{bind } \mathbf{f}_i = \lambda\overline{x_{i,k}}.e_i\left[f_{i,k}\%x_{i,k} \mapsto x_{i,k}\right]; \right)_i ; \texttt{exec}\ (); \langle S'\rangle_c$$

After hoisting, converters can only be the `serial` or `frag`. Its server and client parts are the identity, hence we simply omit them. We also note that $\langle D_s\rangle_s$ differs with D_s only on type annotations and type declarations which are ignored by reduction relations. We note $\widehat{D}_s = \langle D_s\rangle_s\left[\{\{\, e_i \,\}\} \mapsto \texttt{fragment } \mathbf{f_i}\ \overline{x}\right.$

Let us consider the reduction of $(\texttt{bind } \mathbf{f_i} = \lambda\overline{x}_i.\widehat{e_i})_i$. Let us note $e'_i = e_i\left[f_{i,j}\%x_{i,j} \mapsto x_{i,j}\right]$. For any queue ξ, we have the following reduction:

$$\frac{\lambda x_1 \ldots x_m.e'_i, \xi \xrightarrow{\widehat{\rho}_c \mid \widehat{\gamma}_i\to\widehat{\gamma}_i}_{\mathrm{ML_c}} \lambda\overline{x}_i.\widehat{\rho}_c.e'_i, \xi, \langle\rangle}{\forall i, \quad \texttt{bind } \mathbf{f_i} = \lambda\overline{x}_i.e'_i, \xi \xrightarrow{\widehat{\rho}_c \mid \widehat{\gamma}_i\to\widehat{\gamma}_{i+1}}_{\mathrm{ML_c}} \{\}, \xi, \langle\rangle}$$
$$(\texttt{bind } \mathbf{f_i} = \lambda\overline{x}_i.e'_i)_i, \xi \xrightarrow{\widehat{\rho}_c \mid \widehat{\gamma}_1\to\widehat{\gamma}_{n+1}}_{\mathrm{ML_c}} \{\}, \xi, \langle\rangle$$

where $\widehat{\gamma}_1 = \widehat{\gamma}$ and $\widehat{\gamma}_{i+1} = \widehat{\gamma}_i \cup \left\{\mathbf{f_i}{\mapsto}\lambda x_1 \ldots x_m.\widehat{\rho_c}.e'_i\right\}$. Let γ_{bind} be $\left\{\mathbf{f_i}{\mapsto}\lambda x_1 \ldots x_m.\widehat{\rho_c}.e'_i\right\}_i$. We note $\widehat{\gamma}' = \widehat{\gamma}_{n+1} = \widehat{\gamma} \cup \gamma_{bind}$ and $\widehat{\gamma}'_f = \widehat{\gamma}_f \cup \gamma_{bind}$.

Since $(\texttt{bind env } \mathbf{f_i})_i \xrightarrow{\rho_c \mid \gamma \to \gamma'}_c \{\}, \langle\rangle$, we have $\gamma' = \gamma \cup \{\mathbf{f_i}{\mapsto}\rho_c\}_i$. and $\rho_c \simeq^c \widehat{\rho}_c$, we have that $\gamma' \simeq^c \widehat{\gamma}'$. Furthermore, given one of the $\mathbf{f_i}$ in γ_{bind}, each fragment annotated with this $\mathbf{f_i}$ syntactically appear in D_s by uniqueness of the annotation function. This also holds inside functors, since each $\mathbf{f_i}$ will be prefixed by a unique module reference. Hence $FCE(\widehat{\gamma}'_f, D_s)$ and $FCE(\widehat{\gamma}'_f, \rho_s)$.

We now have all the ingredients to uses Corollary 1 on the execution of D_s and μ. This gives us the following reductions:

$$\frac{}{\widehat{D}_s \xrightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \widehat{V}_s, \xi_\bullet, \{\}, \widehat{\theta}_s} \qquad \frac{}{\texttt{exec ()}, \xi_\bullet +\!\!+ \texttt{end} \xrightarrow{\widehat{\rho}_c \mid \widehat{\gamma}' \to \widehat{\gamma}''}_{\mathrm{ML_c}} \varepsilon, [\,], \widehat{\theta}_c}$$

with the following invariants:

$$\widehat{\gamma}'' \simeq^c \gamma'' \qquad FCE(\widehat{\gamma}''_f, V) \qquad \widehat{V} \simeq^s V \qquad \widehat{\theta}_s \simeq^s \theta_s \qquad \widehat{\theta}_c \simeq^c_\zeta \theta_c$$

We remark that $\zeta'' = \zeta$ since no injection took place during a server section and that $\widehat{\gamma}''_f = \widehat{\gamma}'_f$, by definition of the reduction for exec.

We now consider the execution of S'. The following invariants holds:

$$\widehat{\rho}_c \simeq^c_\zeta \rho_c \qquad \widehat{\rho}_s + \widehat{V}_s \simeq^s \rho_s + V_s \qquad \widehat{\gamma}'' \simeq^c \gamma'' \qquad FCE(\widehat{\gamma}''_f, S') \qquad FCE(\widehat{\gamma}''_f, \rho'_s + V_s)$$

By induction on the execution of S' and μ', we have $\langle S'\rangle_s \xrightarrow{\widehat{\rho}_s + \widehat{\mathrm{V}}_s}_{\mathrm{ML_s}} \widehat{V}'_s, \xi'_\bullet, \zeta'_\bullet, \widehat{\theta}'_c$ and $\langle S'\rangle_c, \xi'_\bullet +\!\!+ \xi' \xrightarrow{\widehat{\rho}_c \mid \widehat{\gamma}'' \cup \zeta'_\bullet \to \widehat{\gamma}}$ where

$$\widehat{\gamma}''' \simeq^c \gamma''' \qquad \widehat{V}'_s \simeq^s V'_s \qquad \widehat{\theta}'_s \simeq^s \theta'_s$$
$$FCE(\widehat{\gamma}'''_f, V'_s) \qquad \widehat{V}'_c \simeq^c_{\widehat{\gamma}'} V'_c \qquad \widehat{\theta}'_c \simeq^c_{\widehat{\gamma}'} \theta'_c$$

Finally, we can construct the following executions:

$$\frac{\widehat{D}_s \xrightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \widehat{V}_s, \xi_\bullet, \{\}, \widehat{\theta}_s \qquad \langle S'\rangle_s \xrightarrow{\rho_s + \mathrm{V_s}}_{\mathrm{ML_s}} \widehat{V}'_s, \xi'_\bullet, \zeta'_\bullet, \widehat{\theta}'_s}{\widehat{D}_s; \texttt{end ()}; \langle S'\rangle_s \xrightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \widehat{V}_s + \widehat{V}'_s, \xi_\bullet +\!\!+ \texttt{end} +\!\!+ \xi'_\bullet, \zeta'_\bullet, \widehat{\theta}_s \mathbin{@} \widehat{\theta}'_s}$$

$$\frac{\widehat{\mu}, \xi \xrightarrow{\widehat{\rho}_c \mid \widehat{\gamma} \cup \zeta'_\bullet \to \widehat{\gamma}' \cup \zeta'_\bullet}_{\mathrm{ML_c}} \{\}, \xi, \langle\rangle \qquad \dfrac{\langle S'\rangle_c, \xi'_\bullet +\!\!+ \xi' \xrightarrow{\widehat{\rho}_c \mid \widehat{\gamma}'' \cup \zeta'_\bullet \to \widehat{\gamma}'''}_{\mathrm{ML_c}} V'_c, \xi', \widehat{\theta}'_c}{\texttt{exec ()}; \langle S'\rangle_c, \xi \xrightarrow{\widehat{\rho}_c \mid \widehat{\gamma}' \cup \zeta'_\bullet \to \widehat{\gamma}'''}_{\mathrm{ML_c}} \widehat{V}'_c, \xi', \widehat{\theta}_c \mathbin{@} \widehat{\theta}'_c}}{\widehat{\mu}; \texttt{exec ()}; \langle S'\rangle_c, \xi \xrightarrow{\widehat{\rho}_c \mid \widehat{\gamma} \cup \zeta'_\bullet \to \widehat{\gamma}'''}_{\mathrm{ML_c}} \widehat{V}'_c, \xi', \widehat{\theta}_c \mathbin{@} \widehat{\theta}'_c}$$

where $\xi = \xi_\bullet +\!\!+ \texttt{end} +\!\!+ \xi'_\bullet +\!\!+ \xi'$ and $\widehat{\mu} = (\texttt{bind } \mathbf{f_i} = \lambda\overline{x}_i.e'_i)_i$. We verify the following invariants:

$$\widehat{\gamma}''' \simeq^c \gamma''' \qquad \widehat{V}_s + \widehat{V}'_s \simeq^s V_s + V'_s \qquad \widehat{\theta}_s + \widehat{\theta}'_s \simeq^s \theta_s + \theta'_s$$
$$FCE(\widehat{\gamma}'''_f, V_s + V'_s) \qquad \widehat{V}'_c \simeq^c_{\widehat{\gamma}'''} V'_c \qquad \widehat{\theta}_c + \widehat{\theta}'_c \simeq^c_{\widehat{\gamma}'''} \theta_c + \theta'_c \qquad \blacksquare$$

• Case $S = D_c; S'$ – Client declaration.

We assume that the following executions hold:

$$\frac{D_c \overset{\rho_s}{\Longrightarrow}_{c/s} \overline{D_c}, \varepsilon, \langle\rangle \qquad S' \overset{\rho_s}{\Longrightarrow}_m V'_s, \mu', \theta'_s}{D_c; S' \overset{\rho_s}{\Longrightarrow}_m V'_s, (\overline{D_c}; \mu'), \theta'_s} \qquad \frac{\overline{D_c} \overset{\rho_c \mid \gamma \to \gamma}{\Longrightarrow}_c V_c, \theta_c \qquad \mu' \overset{\rho_c + V_c \mid \gamma \to \gamma'}{\Longrightarrow}_c V'_c, \theta'_c}{\overline{D_c}; \mu' \overset{\rho_c \mid \gamma \to \gamma'}{\Longrightarrow}_c V_c + V'_c, \theta_c \mathbin{@} \theta'_c}$$

Let us note $f_i\%x_i$ the injections in D_c and $\mathbf{x}_i$ the associated fresh variables. Since hoisting has been applied, all the f_i are either `serial` or `frag`. Furthermore, no fragments are executed due to the execution of injections and $\overline{D_c} = D_c[f_i\%x_i \mapsto \rho_s(x_i)]_i$.

We have the following compilations:

$$\langle D_c; S\rangle_s = (\texttt{injection}\ \mathbf{x_i}\ x_i;)_i\ \texttt{end}\ (); \langle S\rangle_s$$
$$\langle D_c; S\rangle_c = \texttt{exec}\ (); D_c[f_i\%x_i \mapsto \mathbf{x}_i]_i\ ; \langle S\rangle_c$$

In the rest of this proof, we note $\widehat{D}_c = D_c[f_i\%x_i \mapsto \mathbf{x}_i]_i$.

We consider the server reduction $D_c \Rightarrow_{c/s} \overline{D_c}$. We know that $\overline{D_c} = D_c[f_i\%x_i \mapsto \downarrow\rho_s(x_i)]_i$. Let us note $v_i = \rho_s(x_i)$. We can build the following $\mathrm{ML_s}$ reduction:

$$\frac{\widehat{\rho}_s(x_i) = \widehat{v}_i}{\dfrac{\forall i.\ \texttt{injection}\ \mathbf{x_i}\ x_i \overset{\widehat{\rho}_s}{\Longrightarrow}_{\mathrm{ML_s}} \varepsilon, [\,], \{\mathbf{x_i} \mapsto \downarrow\widehat{v}_i\}, \langle\rangle}{(\texttt{injection}\ \mathbf{x_i}\ x_i;)_i; \texttt{end}\ () \overset{\widehat{\rho}_s}{\Longrightarrow}_{\mathrm{ML_s}} \varepsilon, \texttt{end}, \{\mathbf{x_i} \mapsto \downarrow\widehat{v}_i\}_i, \langle\rangle}$$

Since $\widehat{\rho}_s \simeq^s \rho_s$, we also have that $\widehat{v}_i \simeq^s v_i$ and $\downarrow\widehat{v}_i \simeq^c_\zeta \downarrow v_i$ for each i. We note $\zeta_\bullet = \{\mathbf{x_i} \mapsto \downarrow\widehat{v}_i\}_i$. By definition of the slicing relation, the $\mathbf{x_i}$ are fresh, hence they are not bound in $\widehat{\gamma}$. We can thus construct the global environment $\widehat{\gamma}' = \widehat{\gamma} \cup \zeta_\bullet$. Since we only extend the part with injection references, we still have that $\gamma \simeq^c \widehat{\gamma}'$.

We now consider the client reduction $\overline{D_c} \overset{\rho_c \mid \gamma \to \gamma}{\Longrightarrow}_c V_c, \theta_c$. We know that $\overline{D_c}$ is equal to $D_c[f_i\%x_i \mapsto \downarrow v_i]_i$, hence the reduction tree contains for each i a reduction $\downarrow v_i \overset{\mid \gamma \to \gamma}{\Longrightarrow}_c \downarrow v_i, \langle\rangle$. To obtain a reduction of $\widehat{D}_c = D_c[f_i\%x_i \mapsto \mathbf{x}_i]_i$, we simply substitute each of these subreduction by one of the form $\mathbf{x_i}, \xi \overset{\mid \widehat{\gamma}' \to \widehat{\gamma}'}{\Longrightarrow}_{\mathrm{ML_c}} \downarrow\widehat{v}_c, \xi, \langle\rangle$. for any queue ξ. By Lemma 1, we can build the following reduction:

$$\widehat{D}_c, \xi \overset{\widehat{\rho}_c \mid \widehat{\gamma}' \to \widehat{\gamma}'}{\Longrightarrow}_{\mathrm{ML_c}} \widehat{V}_c, \xi, \widehat{\theta}_c$$

where $\widehat{V}_c \simeq^c_{\zeta'} V_c$ and $\widehat{\theta}_c \simeq^c_{\zeta'} \theta_c$, for any queue ξ.

We now consider the execution of S'. We have the following properties:

$$\widehat{\rho}_c + \widehat{V}_c \simeq^c_{\zeta'} \rho_c + V_c \qquad \widehat{\rho}_s \simeq^s \rho_s \qquad \widehat{\gamma}' \simeq^c \gamma \qquad FCE(\widehat{\gamma}_f, S') \qquad FCE(\widehat{\gamma}_f, \rho_s)$$

By induction on the execution of S' and μ', we have $\langle S'\rangle_c \overset{\widehat{\rho}_c + \widehat{V}_c}{\Longrightarrow}_{\mathrm{ML_s}} \widehat{V}'_s, \xi'_\bullet, \zeta'_\bullet, \widehat{\theta}'_c$ and $\langle S'\rangle_s, \xi'_\bullet +\!\!+ \xi' \overset{\widehat{\rho}_s \mid \zeta'_\bullet \cup \widehat{\gamma}' \to \widehat{\gamma}'}{\Longrightarrow}$ where

$$\widehat{\gamma}'' \simeq^c \gamma' \qquad \widehat{V}'_s \simeq^s_{\zeta''} V'_s \qquad \widehat{\theta}'_s \simeq^s_{\zeta''} \theta'_s$$
$$FCE(\widehat{\gamma}''_f, V'_s) \qquad \widehat{V}'_c \simeq^c_{\zeta''} V'_c \qquad \widehat{\theta}'_c \simeq^c_{\zeta''} \theta'_c$$

Finally, we can build the following derivations:

$$\frac{\forall i,\ \texttt{injection}\ \mathbf{x_i}\ x_i \xRightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \varepsilon, [\,], \zeta_\bullet, \langle\rangle \qquad \langle S'\rangle_s \xRightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \widehat{V}'_s, \xi'_\bullet, \zeta'_\bullet, \widehat{\theta}'_s}{\left(\texttt{injection}\ \mathbf{x_i}\ (f_i^s\ x_i);\right)_i \texttt{ end }(); \langle S'\rangle_s \xRightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \widehat{V}'_s, \texttt{end} +\!\!+ \xi'_\bullet, \zeta_\bullet \cup \zeta'_\bullet, \widehat{\theta}'_s}$$

$$\frac{\dfrac{\widehat{D}_c, \xi'_\bullet +\!\!+ \xi' \xRightarrow{\widehat{\rho}_c \mid \widehat{\gamma}' \cup \zeta' \bullet \to \widehat{\gamma}' \cup \zeta' \bullet}_{\mathrm{ML_c}} \widehat{V}_c, \xi'_\bullet +\!\!+ \xi', \widehat{\theta}'_c \qquad \langle S'\rangle_c, \xi'_\bullet +\!\!+ \xi' \xRightarrow{\widehat{\rho}_c + \widehat{\mathrm{V}}_c \mid \widehat{\gamma}' \cup \zeta' \bullet \to \widehat{\gamma}''}_{\mathrm{ML_c}} \widehat{V}'_c, \xi', \widehat{\theta}'_c}{\widehat{D}_c; \langle S'\rangle_c, \xi'_\bullet +\!\!+ \xi' \xRightarrow{\widehat{\rho}_c \mid \widehat{\gamma}' \cup \zeta' \bullet \to \widehat{\gamma}''}_{\mathrm{ML_c}} \widehat{V}_c + \widehat{V}'_c, \xi', \widehat{\theta}_c \mathbin{@} \widehat{\theta}'_c}}{\texttt{exec }(); \widehat{D}_c; \langle S'\rangle_c, \texttt{end} +\!\!+ \xi'_\bullet +\!\!+ \xi' \xRightarrow{\widehat{\rho}_c \mid \widehat{\gamma} \cup \zeta_\bullet \cup \zeta' \bullet \to \widehat{\gamma}''}_{\mathrm{ML_c}} \widehat{V}_c + \widehat{V}'_c, \xi', \widehat{\theta}_c \mathbin{@} \widehat{\theta}'_c}$$

We verify the following invariants:

$$\begin{array}{ccc} \widehat{\gamma}'' \simeq^c \gamma' & \widehat{V}_s \simeq^s V_s & \widehat{\theta}_s \simeq^s \theta_s \\ FCE(\widehat{\gamma}''_f, V_s + V'_s) & \widehat{V}_c + \widehat{V}'_c \simeq^c_{\widehat{\gamma}''} V_c + V'_c & \widehat{\theta}_c + \widehat{\theta}'_c \simeq^c_{\widehat{\gamma}''} \theta_c + \theta'_c \end{array} \qquad \blacksquare$$

• Case $\texttt{module}_m\ X = M; S'$ – Declaration of a mixed module.
We assume that the following executions hold:

$$\frac{\dfrac{M \xRightarrow{\rho_s}_m V_s, M^c, \mu, \theta_s}{\texttt{module}_m\ X = M \xRightarrow{\rho_s}_m \{X \mapsto V_s\}, \texttt{module } X = M^c; \mu, \theta_s \qquad S' \xRightarrow{\rho_s + \{\mathrm{X} \mapsto \mathrm{V_s}\}}_m V'_s, \mu', \theta'_s}}{\texttt{module}_m\ X = M; S' \xRightarrow{\rho_s}_m \{X \mapsto V_s\} + V'_s, (\mu; \texttt{module } X = M^c; \mu'), \theta_s \mathbin{@} \theta'_s}$$

$$\frac{\mu \xRightarrow{\rho_c \mid \gamma \to \gamma'}_c \{\}, \theta_c \qquad \dfrac{M^c \xRightarrow{\rho_c \mid \gamma' \to \gamma''}_c V_c, \theta'_c}{\texttt{module } X = M^c \xRightarrow{\rho_c \mid \gamma' \to \gamma''}_c \{X \mapsto V_c\}, \theta'_c} \qquad \mu' \xRightarrow{\rho_c + \{\mathrm{X} \mapsto \mathrm{V_c}\} \mid \gamma'' \to \gamma'''}_c V'_c, \theta''_c}{\mu; \texttt{module } X = M^c; \mu' \xRightarrow{\rho_c \mid \gamma \to \gamma'''}_c \{X \mapsto V_c\} + V'_c, \theta_c \mathbin{@} \theta'_c \mathbin{@} \theta''_c}$$

Let us assumes that we can build the following reductions

$$\frac{}{\langle \texttt{module}_m\ X = M\rangle_s \xRightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \left\{X \mapsto \widehat{V}_s\right\}, \xi_\bullet, \zeta, \widehat{\theta}_s}$$

$$\frac{}{\langle \texttt{module}_m\ X = M\rangle_c, \xi_\bullet +\!\!+ \xi \xRightarrow{\widehat{\rho}_c \mid \zeta \cup \widehat{\gamma} \to \widehat{\gamma}'}_{\mathrm{ML_c}} \left\{X \mapsto \widehat{V}_c\right\}, \xi, \widehat{\theta}_c}$$

for any ξ, and that the following invariants hold:

$$\begin{array}{ccc} \widehat{\gamma}' \simeq^c \gamma'' & \widehat{V}_s \simeq^s V_s & \widehat{\theta}_s \simeq^s \theta_s \\ FCE(\widehat{\gamma}'_f, V_s) & \widehat{V}_c \simeq^c_{\widehat{\gamma}''} V_c & \widehat{\theta}_c \simeq^c_{\widehat{\gamma}'} \theta_c + \theta'_c \end{array}$$

By induction on the execution of S' and μ', we can build the following reduction: $\langle S'\rangle_s \xRightarrow{\widehat{\rho}_s + \left\{\mathrm{X} \mapsto \widehat{\mathrm{V}}_s\right\}}_{\mathrm{ML_s}} V'_s, \xi'_\bullet, \zeta'$ and $\langle S'\rangle_c, \xi'_\bullet +\!\!+ \xi \xRightarrow{\widehat{\rho}_c + \left\{\mathrm{X} \mapsto \widehat{\mathrm{V}}_c\right\} \mid \zeta' \cup \widehat{\gamma}' \to \widehat{\gamma}''}_{\mathrm{ML_c}} V'_c, \xi, \theta'_c$, which allows us to conclude.

To build the compiled reduction, we will operate by case analysis over M.

• Subcase $M = \texttt{struct}\ S\ \texttt{end}_{\mathbf{X}}$ – Declaration of a mixed structure.
We have $\mu = \texttt{bind}\ \mathbf{X} = (\texttt{struct}\ \mu_0\ \texttt{end})$ and $M^c = \mathbf{X}$ with the following reductions:

$$\frac{S \overset{\rho_s}{\Longrightarrow}_m V_s, \mu, \theta_s}{\texttt{struct}\ S\ \texttt{end} \overset{\rho_s}{\Longrightarrow}_m V_s + \{\texttt{Dyn}\mapsto\mathbf{X}\}, \mathbf{X}, \mu, \theta_s}$$

$$\frac{\dfrac{}{\mu_0 \overset{\rho_c \mid \gamma\to\gamma'}{\Longrightarrow}_c V_c, \theta_c} \qquad \dfrac{\gamma''(\mathbf{X}) = V_c}{\mathbf{X} \overset{\rho_c \mid \gamma''\to\gamma''}{\Longrightarrow}_c V_c, \langle\rangle}}{(\texttt{bind}\ \mathbf{X} = \texttt{struct}\ \mu_0\ \texttt{end};\ \texttt{module}\ X = \mathbf{X}) \overset{\rho_c \mid \gamma\to\gamma''}{\Longrightarrow}_c, \theta_c}$$

where $\gamma'' = \gamma' \cup \{\mathbf{X}\mapsto V_c\}$.

We have the following compilations:

$$\langle \texttt{module}_m\ X = \texttt{struct}\ S\ \texttt{end}_{\mathbf{X}} \rangle_s = \begin{array}{l}\texttt{module}\ X = \texttt{struct}\\ \quad \texttt{module}\ \texttt{Dyn} = \mathbf{X};\\ \quad \langle S \rangle_s\\ \texttt{end}\end{array}$$

$$\langle \texttt{module}_m\ X = \texttt{struct}\ S\ \texttt{end}_{\mathbf{X}} \rangle_c = \begin{array}{l}\texttt{bind}_m\ \mathbf{X} = \texttt{struct}\ \langle S \rangle_c\ \texttt{end};\\ \texttt{module}\ X = \mathbf{X};\end{array}$$

By induction on the execution of S and μ, we have $\langle S \rangle_s \overset{\widehat{\rho}_s}{\Longrightarrow}_{\mathrm{ML_s}} \widehat{V}_s, \xi_\bullet, \zeta_\bullet, \widehat{\theta}_s$ and $\langle S \rangle_c, \xi_\bullet +\!\!+ \xi \overset{\widehat{\rho}_c \mid \widehat{\gamma}\cup\zeta_\bullet\to\widehat{\gamma}'}{\Longrightarrow}_{\mathrm{ML_c}} \widehat{V}$
with the following invariants:

$$\begin{array}{lll} \widehat{\gamma}' \simeq^c \gamma' & \widehat{V}_s \simeq^s V_s & \widehat{\theta}_s \simeq^s \theta_s \\ FCE(\widehat{\gamma}'_f, V_s) & \widehat{V}_c \simeq^c_{\widehat{\gamma}'} V_c & \widehat{\theta}_c \simeq^c_{\widehat{\gamma}'} \theta_c \end{array}$$

We can then build the following executions:

$$\frac{\langle S \rangle_s \overset{\widehat{\rho}_s}{\Longrightarrow}_{\mathrm{ML_s}} \widehat{V}_s, \xi_\bullet, \zeta_\bullet, \widehat{\theta}_s}{\langle \texttt{module}_m\ X = \texttt{struct}\ S\ \texttt{end}_{\mathbf{X}} \rangle_s \overset{\widehat{\rho}_s}{\Longrightarrow}_{\mathrm{ML_s}} \left\{X\mapsto\{\texttt{Dyn}\mapsto\mathbf{X}\} + \widehat{V}_s\right\}, \xi_\bullet, \zeta_\bullet, \widehat{\theta}_s}$$

$$\frac{\dfrac{\langle S \rangle_c, \xi_\bullet +\!\!+ \xi \overset{\widehat{\rho}_c \mid \widehat{\gamma}\cup\zeta_\bullet\to\widehat{\gamma}'}{\Longrightarrow}_{\mathrm{ML_c}} \widehat{V}_c, \xi, \widehat{\theta}_c}{\texttt{module}\ X = \texttt{struct}\ \langle S \rangle_c\ \texttt{end}, \xi_\bullet +\!\!+ \xi \overset{\widehat{\rho}_c \mid \widehat{\gamma}\cup\zeta_\bullet\to\widehat{\gamma}'}{\Longrightarrow}_{\mathrm{ML_c}} \left\{X\mapsto\widehat{V}_c\right\}, \xi, \widehat{\theta}_c}}{\langle \texttt{module}_m\ X = \texttt{struct}\ S\ \texttt{end} \rangle_c, \xi_\bullet +\!\!+ \xi \overset{\widehat{\rho}_c \mid \widehat{\gamma}\cup\zeta_\bullet\to\widehat{\gamma}''}{\Longrightarrow}_{\mathrm{ML_c}} \left\{X\mapsto\widehat{V}_c\right\}, \xi, \widehat{\theta}_c}$$

Where $\widehat{\gamma}'' = \widehat{\gamma}' \cup \left\{\mathbf{X}\mapsto\widehat{V}_c\right\}$. We verify the following invariants:

$$\begin{array}{lll} \widehat{\gamma}'' \simeq^c \gamma' & \widehat{V}_s + \{\texttt{Dyn}\mapsto\mathbf{X}\} \simeq^s V_s + \{\texttt{Dyn}\mapsto\mathbf{X}\} & \widehat{\theta}_s \simeq^s \theta_s \\ FCE(\widehat{\gamma}''_f, V_s) & \widehat{V}_c \simeq^c_{\widehat{\gamma}'} V_c & \widehat{\theta}_c \simeq^c_{\widehat{\gamma}'} \theta_c \end{array}$$

which concludes. ■

• Subcase $M = \texttt{functor}(X_i : \mathcal{M}_i)_i \texttt{struct}\ S\ \texttt{end}_{\mathbf{F}}$ – Declaration of a mixed functor.

In this case, we have the client program μ = `bind env` $\mathbf{F}$ and the module expression M^c = `functor`$(X_i : \mathcal{M}_i)_i$`struct` $S|_c$ `end`. The following reductions hold:

$$\frac{}{\begin{pmatrix}\texttt{module}_m\ F(X_i:\mathcal{M}_i)_i = \texttt{struct}\\ \quad S\\ \texttt{end}_{\mathbf{F}}\end{pmatrix} \overset{\rho_s}{\Longrightarrow}_m \{F \mapsto V_s\}, \begin{pmatrix}\texttt{bind env}\ \mathbf{F}\\ \texttt{module}_m\ F(X_i:\mathcal{M}_i)_i = \texttt{struct}\\ \quad S|_c\\ \texttt{end}\end{pmatrix}, \langle\rangle}$$

$$\frac{}{\begin{pmatrix}\texttt{bind env}\ \mathbf{F}\\ \texttt{module}_m\ F(X_i:\mathcal{M}_i)_i = \texttt{struct}\\ \quad S|_c\\ \texttt{end}\end{pmatrix} \overset{\rho_c \mid \gamma \to \gamma'}{\Longrightarrow}_c \{F \mapsto V_c\}, \langle\rangle}$$

Where $\gamma' = \gamma \cup \{\mathbf{F} \mapsto \rho_c\}$ and the following values:

$$V_s = \texttt{functor}(\rho_s)(X_i:\mathcal{M}_i)_i\texttt{struct}\ S\ \texttt{end}$$
$$V_c = \texttt{functor}(\rho_c)(X_i:\mathcal{M}_i)_i\texttt{struct}\ S|_c\ \texttt{end}$$

We recall that by hoisting, the body of the functors contains no injection, hence we don't need to evaluate server code in the client part.

We have the following compilations:

$$\left\langle \begin{array}{l}\texttt{module}_m\ F(X_i:\mathcal{M}_i)_i = \texttt{struct}\\ \quad S\\ \texttt{end}_{\mathbf{F}}\end{array} \right\rangle_s = \begin{array}{l}\texttt{module}\ F(X_i:\langle\mathcal{M}_i\rangle_s)_i = \texttt{struct}\\ \quad \texttt{module Dyn} = \texttt{fragment}_m\ \mathbf{F}\ (X_i.\texttt{Dyn})_i;\\ \quad \langle S\rangle_s\\ \texttt{end}\end{array}$$

$$\left\langle \begin{array}{l}\texttt{module}_m\ F(X_i:\mathcal{M}_i)_i = \texttt{struct}\\ \quad S\\ \texttt{end}_{\mathbf{F}}\end{array} \right\rangle_c = \begin{array}{l}\texttt{bind}_m\ \mathbf{F}(X_i:\langle\mathcal{M}_i\rangle_c)_i = \texttt{struct}\ \langle S\rangle_c\ \texttt{end;}\\ \texttt{module}\ F(X_i:\langle\mathcal{M}_i\rangle_c)_i = \texttt{struct}\ S|_c\ \texttt{end;}\end{array}$$

We trivially have the following execution:

$$\frac{}{\left\langle \begin{array}{l}\texttt{module}_m\ F(X_i:\mathcal{M}_i)_i = \texttt{struct}\\ \quad S\\ \texttt{end}_{\mathbf{F}}\end{array} \right\rangle_s \overset{\widehat{\rho}_s}{\Longrightarrow}_{\mathrm{ML_s}} \left\{F \mapsto \widehat{V}_s\right\}, \xi_\bullet, \{\}, \langle\rangle}$$

$$\frac{}{\left\langle \begin{array}{l}\texttt{module}_m\ F(X_i:\mathcal{M}_i)_i = \texttt{struct}\\ \quad S\\ \texttt{end}_{\mathbf{F}}\end{array} \right\rangle_c, \xi \overset{\widehat{\rho}_c \mid \widehat{\gamma} \to \widehat{\gamma}'}{\Longrightarrow}_{\mathrm{ML_c}} \left\{F \mapsto \widehat{V}_c\right\}, \xi, \langle\rangle}$$

Where $\gamma' = \gamma \cup \left\{\mathbf{F} \mapsto \widehat{V}_{\mathbf{F}}\right\}$ with the following values:

$$\widehat{V}_s = \texttt{functor}(\widehat{\rho}_s)(X_i:\mathcal{M}_i)_i\texttt{struct}\ \langle S\rangle_s\ \texttt{end}$$
$$\widehat{V}_c = \texttt{functor}(\widehat{\rho}_c)(X_i:\mathcal{M}_i)_i\texttt{struct}\ S|_c\ \texttt{end}$$
$$\widehat{V}_{\mathbf{F}} = \texttt{functor}(\widehat{\rho}_c)(X_i:\mathcal{M}_i)_i\texttt{struct}\ \langle S\rangle_c\ \texttt{end}$$

We now need to show that the invariants still hold. We easily have that $\widehat{V}_c \simeq^c_{\widehat{\gamma}} V_c$. By definition of equivalence over mixed functors, we have $\widehat{V}_s \simeq^s V_s$. Indeed, the body of the functor in $\widehat{V}_s$ is the server compilation of the body of the mixed functor V_s and the captured environments corresponds. Finally, we have that the body of $\widehat{V}_\mathbf{F}$ is the client compilation of the body of the mixed functors and that the capture environment corresponds to $\gamma'(\mathbf{F})$. Thus we get that $FCE(\widehat{\gamma}'_f, V_s)$. By definition of the annotation function, the reference $\mathbf{F}$ could not have appeared on a previously executed structure, hence we still have that $FCE(\widehat{\gamma}'_f, \widehat{\rho}_s)$.

Hence, all the following invariants are respected, which concludes.

$$\begin{array}{ll} \widehat{\gamma}' \simeq^c \gamma' & \widehat{V}_s \simeq^s V_s \\ FCE(\widehat{\gamma}'_f, V_s) & \widehat{V}_c \simeq^c_{\widehat{\gamma}''} V_c \end{array}$$ ■

Otherwise, M is a module expression. By definition of slicability, M does not syntactically contain any structure. In the general case, we should proceed by induction over module expressions. We will simply present the case of a mixed functor application where the functor returns a mixed structure.

We consider $M = F(X_1)\dots(X_n)$. We have $M^c = F(X_1)\dots(X_n)$ with the following executions:

$$\frac{\begin{array}{l} F \xRightarrow{\rho_s}_m \texttt{functor}_m(\rho'_s)(Y_i : \mathcal{M}_i)_i \texttt{struct}\ S\ \texttt{end}_\mathbf{F}, \varepsilon, \langle\rangle \\ X_i \xRightarrow{\rho_s}_m V^s_i, \varepsilon, \langle\rangle \qquad V^s_i(\texttt{Dyn}) = \mathbf{R_i} \qquad \mathbf{R}\ \text{fresh} \qquad S[\mathbf{f_i} \mapsto \mathbf{R}.\mathbf{f_i}]_i \xRightarrow{\rho'_s + \{Y_i \mapsto V^s_i\}_i}_m V, \mu_0, \theta \end{array}}{F(X_1)\dots(X_n) \xRightarrow{\rho_s}_m V_s + \{\texttt{Dyn} \mapsto \mathbf{R}\}, \mu = \begin{pmatrix} \texttt{bind}\ \mathbf{R} = \texttt{struct} \\ \quad (\texttt{module}\ Y_i = \mathbf{R_i};)_i \\ \quad \mu_0 \\ \texttt{end with}\ \mathbf{F} \end{pmatrix}, \theta}$$

$$\texttt{module}_m\ X = F(X_1)\dots(X_n) \xRightarrow{\rho}_m V_s, (\mu; \texttt{module}\ X = F(X_1)\dots(X_n)), \theta_s$$

$$\frac{\gamma(\mathbf{F}) = \rho_\mathbf{F} \qquad \gamma(\mathbf{R_i}) = V^c_i \qquad \mu_0 \xRightarrow{\rho_\mathbf{F} + \{Y_i \mapsto V^c_i\}_i \mid \gamma \to \gamma'}_c V_c, \theta_c}{\begin{pmatrix} \texttt{bind}\ \mathbf{R} = \texttt{struct} \\ \quad (\texttt{module}\ Y_i = \mathbf{R_i};)_i \\ \quad \mu_0 \\ \texttt{end with}\ \mathbf{F} \end{pmatrix} \xRightarrow{\rho_c \mid \gamma \to \gamma' \cup \{\mathbf{R} \mapsto V_c\}}_c \varepsilon, \theta_c}$$

$$\frac{\begin{array}{l} F \xRightarrow{\rho_s \mid \cdots \to \cdots}_c \texttt{functor}(\rho'_c)(Y_i : \mathcal{M}_i)_i \texttt{struct}\ S_c\ \texttt{end}, \langle\rangle \\ X_i \xRightarrow{\rho_c \mid \cdots \to \cdots}_c V_i, \langle\rangle \qquad S_c \xRightarrow{\rho'_c + \{Y_i \mapsto V_i\}_i \mid \gamma' \cup \{\mathbf{R} \mapsto V_c\} \to \gamma''}_c V'_c, \theta'_c \end{array}}{\texttt{module}\ X = F(X_1)\dots(X_n) \xRightarrow{\rho_c \mid \gamma' \cup \{\mathbf{R} \mapsto V_c\} \to \gamma''}_c \{X \mapsto V'_c\}, \theta'_c}$$

We note V_F the value of F in γ_s, which is $\texttt{functor}(\rho'_s)(Y_i : \mathcal{M}_i)_i \texttt{struct}\ S\ \texttt{end}$.

We have the following compilations:

$$\langle \texttt{module}_m\ X = F(X_1)\dots(X_n) \rangle_s = \begin{array}{l} \texttt{module}\ X = F(X_1)\dots(X_n); \\ \texttt{end}\ (); \end{array}$$

$$\langle \texttt{module}_m\ X = F(X_1)\dots(X_n) \rangle_c = \begin{array}{l} \texttt{module}\ X = F(X_1)\dots(X_n); \\ \texttt{exec}\ (); \end{array}$$

Let us now look at the execution of the server application $F(X_1)\dots(X_n)$. By hypothesis, V_F is a mixed functor. By equivalence, we know that $\widehat{\rho}_s(F) = \widehat{V}_F$ where $\widehat{V}_F \simeq^s V_F$. By definition of the equivalence on server values, $\widehat{V}_F$ is of the following shape:

$$\widehat{V}_F = \begin{pmatrix} \texttt{functor}(\widehat{\rho}'_s)(Y_i : \mathcal{M}_i)_i\texttt{struct} \\ \quad \texttt{module Dyn} = \texttt{fragment}_m\ \mathbf{F}\ (Y_i.\texttt{Dyn})_i; \\ \quad \langle S \rangle_s \\ \texttt{end} \end{pmatrix}$$

where $\widehat{\rho}'_s \simeq^s_{\widehat{\gamma}} \rho'_s$. For each i we note $\widehat{V}^s_i = \widehat{\rho}_s(X_i)$. By equivalence, we have $\widehat{V}^s_i \simeq^s V^s_i$.

Furthermore, since $\widehat{\gamma} \simeq^c \gamma$ and by hypothesis, $\mathbf{F}$ is also bound in $\widehat{\gamma}$. We note $V_\mathbf{F}$ the corresponding value. Since $FCE(\widehat{\gamma}, V_c)$ (via ρ_s), then $\widehat{V}_\mathbf{F}$ is of the following shape:

$$\widehat{V}_\mathbf{F} = \begin{pmatrix} \texttt{functor}(\widehat{\rho}_\mathbf{F})(Y_i : \mathcal{M}_i)_i\texttt{struct} \\ \quad \langle S \rangle_c \\ \texttt{end} \end{pmatrix}$$

where $\widehat{\rho}_\mathbf{F} \simeq^c_{\widehat{\gamma}} \rho_\mathbf{F} = \gamma(\mathbf{F})$. Additionally, for each i we note $V^c_i = \widehat{\gamma}(\mathbf{R_i})$. By equivalence, we have $\widehat{V}^c_i \simeq^s V^c_i$.

We can now proceed by induction on S and μ_0 in the environment $\widehat{\gamma}$, $\widehat{\rho}_c \cup \{Y_i{\mapsto}\mathbf{R_i}\}$ and $\widehat{\rho}_s \cup \{\texttt{Dyn}{\mapsto}\mathbf{R}\} \cup \left\{Y_i{\mapsto}\widehat{V}^s_i\right\}_i$. We obtain the following reductions:

$\langle S \rangle_s \xrightarrow{\widehat{\rho}'_s \cup \left\{Y_i \mapsto \widehat{V}^s_i\right\}_i}_{\text{ML}_s} \xi_\bullet, \zeta_\bullet, \widehat{\theta}_s$, and $\langle S \rangle_c, \xi_\bullet +\!\!+ \xi \xrightarrow{\widehat{\rho}_c \cup \{Y_i \mapsto \mathbf{R_i}\}_i \mid \zeta_\bullet \cup \widehat{\gamma} \to \widehat{\gamma}'}_{\text{ML}_c} \widehat{V}_c, \xi, \widehat{\theta}_c$
with the usual invariants. We can now build the following executions:

$$\frac{\texttt{fragment}_m\ \mathbf{F}\ (Y_i.\texttt{Dyn})_i \xrightarrow{\widehat{\rho}'_s \cup \left\{Y_i \mapsto \widehat{V}^s_i\right\}_i}_{\text{ML}_s} \mathbf{R}, \xi_\mathbf{R}, \varepsilon, \langle\rangle \qquad \widehat{\rho}_s(F) = \widehat{V}_f \qquad \widehat{\rho}_s(X_i) = \widehat{V}^s_i \qquad \langle S \rangle_s \xrightarrow{\widehat{\rho}'_s \cup \{\texttt{Dyn} \mapsto \mathbf{R}\} \cup \left\{Y_i \mapsto \widehat{V}^s_i\right\}_i}_{\text{ML}_s} \xi_\bullet, \zeta_\bullet, \widehat{\theta}_s,}{\texttt{module}\ X = F(X_1)\dots(X_n); \texttt{end}\ () \xrightarrow{\widehat{\rho}_s}_{\text{ML}_s} \left\{X \mapsto \widehat{V}_s\right\}, \xi_\mathbf{R} +\!\!+ \xi_\bullet +\!\!+ \texttt{end}, \zeta_\bullet, \widehat{\theta}_s}$$

$$\frac{\widehat{\gamma}(\mathbf{F}) = \widehat{V}_\mathbf{F} \qquad \langle S \rangle_c, \xi_\bullet +\!\!+ \texttt{end} +\!\!+ \xi \xrightarrow{\widehat{\rho}_\mathbf{F} \cup \{Y_i \mapsto \mathbf{R_i}\}_i \mid \zeta_\bullet \cup \widehat{\gamma} \to \widehat{\gamma}'}_{\text{ML}_c} \widehat{V}_c, \texttt{end} +\!\!+ \xi, \widehat{\theta}_c}{\texttt{exec}\ (), \xi_\mathbf{R} +\!\!+ \xi_\bullet +\!\!+ \texttt{end} +\!\!+ \xi \xrightarrow{\widehat{\rho}_c \mid \zeta_\bullet \cup \widehat{\gamma} \to \widehat{\gamma}' \cup \left\{\mathbf{R} \mapsto \widehat{V}_c\right\}}_{\text{ML}_c} \varepsilon, \xi, \widehat{\theta}_c}$$

where $\xi_\mathbf{R} = \{\mathbf{R}{\mapsto}\mathbf{F}(\mathbf{R_1})\dots(\mathbf{R_n})\}$. We respect the following invariants:

$\widehat{\gamma}' \simeq^c \gamma'$	$\widehat{V}_s \simeq^s V_s$	$\widehat{\theta}_s \simeq^s \theta_s$
$FCE(\widehat{\gamma}'_f, V_s)$	$\widehat{V}_c \simeq^c_{\widehat{\gamma}'} V_c$	$\widehat{\theta}_c \simeq^c_{\widehat{\gamma}'} \theta_c$

Let us now consider the client application. Since $\rho_c \simeq^c_{\widehat{\gamma}} \widehat{\rho}_c$, we have that the body of the functor F is equivalent. We can thus build the following reduction:

$$\frac{\widehat{\rho}_c(F) = \texttt{functor}(\widehat{\rho}'_c)(Y_i : \mathcal{M}_i)_i\texttt{struct}\ \widehat{S}_c\ \texttt{end} \qquad \widehat{\rho}_c(X_i) = \widehat{V}^c_i \qquad \widehat{S}_c \xrightarrow{\widehat{\rho}'_c + \left\{Y_i \mapsto \widehat{V}^c_i\right\}_i \mid \widehat{\gamma}' \cup \left\{\mathbf{R} \mapsto \widehat{V}_c\right\} \to \widehat{\gamma}''}_{\text{ML}_c} \widehat{V}'_c, \widehat{\theta}'_c}{\texttt{module}\ X = F(X_1)\dots(X_n), \xi \xrightarrow{\widehat{\rho}_c \mid \widehat{\gamma}' \cup \left\{\mathbf{R} \mapsto \widehat{V}_c\right\} \to \widehat{\gamma}''}_{\text{ML}_c} \left\{X \mapsto \widehat{V}'_c\right\}, \xi, \widehat{\theta}'_c}$$

By equivalence of F and X_i in ρ_c and $\widehat{\rho}_c$, we have that $\widehat{\gamma}' \simeq^c \gamma'$, $\widehat{V}'_c \simeq^c_{\widehat{\gamma}'} V'_c$ and $\widehat{\theta}_c \simeq^c_{\widehat{\gamma}'} \theta_c$.

We already built the reduction for the compiled server program. We can now build the compiled client program:

$$\frac{\begin{array}{l}\texttt{exec}\ (), \{\mathbf{R}\mapsto\mathbf{F}(\mathbf{R_1})\ldots(\mathbf{R_n})\}{+\!\!+}\xi_\bullet{+\!\!+}\texttt{end}{+\!\!+}\xi \xrightarrow{\widehat{\rho}_c \mid \zeta_\bullet\cup\widehat{\gamma}\to\widehat{\gamma}'\cup\left\{\mathbf{R}\mapsto\widehat{\mathrm{V}}_c\right\}}_{\mathrm{ML_c}} \varepsilon, \xi, \widehat{\theta}_c \\ \texttt{module}\ X = F(X_1)\ldots(X_n), \xi \xrightarrow{\widehat{\rho}_c \mid \widehat{\gamma}'\cup\left\{\mathbf{R}\mapsto\widehat{\mathrm{V}}_c\right\}\to\widehat{\gamma}''}_{\mathrm{ML_c}} \left\{X\mapsto\widehat{V}'_c\right\}, \xi, \widehat{\theta}'_c\end{array}}{\langle\texttt{module}_m\ X = F(X_1)\ldots(X_n)\rangle_c, \xi_{\mathbf{R}}{+\!\!+}\xi_\bullet{+\!\!+}\texttt{end}{+\!\!+}\xi \xrightarrow{\widehat{\rho}_c \mid \zeta_\bullet\cup\widehat{\gamma}\to\widehat{\gamma}''}_{\mathrm{ML_c}} \left\{X\mapsto\widehat{V}'_c\right\}, \xi, \widehat{\theta}_c \mathbin{@} \widehat{\theta}'_c}$$

where the invariants still hold. This concludes. ■

□

B.5 Proof of the main theorem

Finally, we prove Theorem 4. This is a direct consequence of Lemma 3.

Proof of Theorem 4. We have that $P \overset{\{\}}{\Longrightarrow} v, \theta$. By definition of an Eliom$_\varepsilon$ program execution, we can decompose this rule as following:

$$\frac{P \overset{\{\}}{\Longrightarrow}_m (), \mu, \theta_s \qquad \mu \xrightarrow{\{\} \mid \varepsilon\to\gamma}_c v, \theta_c}{P \overset{\{\}}{\Longrightarrow} v, \theta_s \mathbin{@} \theta_c}$$

We trivially have the following invariants:

$$\{\} \simeq^c_{\{\}} \{\} \qquad \{\} \simeq^s \{\} \qquad \widehat{\gamma} \simeq^c \gamma \qquad FCE(\{\}, P) \qquad FCE(\{\}, \{\})$$

which allow us to apply Lemma 3 and conclude. □