université PARIS DIDEROT PARIS 7 | USPC Université Sorbonne Paris Cité

INSTITUT DE RECHERCHE EN INFORMATIQUE FONDAMENTALE

Thèse de doctorat de l'Université Sorbonne Paris Cité
Préparée à l'Université Paris Diderot au Laboratoire IRIF

Ecole Doctorale 386 — Science Mathématiques De Paris Centre

Tierless Web programming in ML

par

Gabriel Radanne

Dirigée par
Roberto Di Cosmo et Jérôme Vouillon

Thèse soutenue publiquement le 14 Novembre 2017 devant le jury constitué de

Roberto Di Cosmo	Directeur de thèse
Jérôme Vouillon	Co-Directeur de thèse
Koen Claessen	Rapporteur
Jacques Garrigue	Rapporteur
Coen De Roover	Examinateur
Xavier Leroy	Examinateur
Manuel Serrano	Examinateur
Jeremy Yallop	Examinateur

Abstract ELIOM is a dialect of OCAML for Web programming in which server and client pieces of code can be mixed in the same file using syntactic annotations. This allows to build a whole application as a single distributed program, in which it is possible to define in a composable way reusable widgets with both server and client behaviors.

ELIOM is type-safe, as it ensures that communications are well-behaved through novel language constructs that match the specificity of Web programming. ELIOM is also efficient, it provides static slicing which separates client and server parts at compile time and avoids back-and-forth communications between the client and the server. Finally, ELIOM supports modularity and encapsulation thanks to an extension of the OCAML module system featuring tierless annotations that specify whether some definitions should be on the server, on the client, or both.

This thesis presents the design, the formalization and the implementation of the ELIOM language.

Résumé ELIOM est un dialecte d'OCAML pour la programmation Web qui permet, à l'aide d'annotations syntaxiques, de déclarer code client et code serveur dans un même fichier. Ceci permet de construire une application complète comme un unique programme distribué dans lequel il est possible de définir des widgets aisément composables avec des comportements à la fois client et serveur.

ELIOM assure un bon comportement des communications grâce à un système de type et de nouvelles constructions adaptés à la programmation Web. De plus, ELIOM est efficace : un découpage statique sépare les parties client et serveur durant la compilation et évite de trop nombreuses communications entre le client et le serveur. Enfin, ELIOM supporte la modularité et l'encapsulation grâce à une extension du système de module d'OCAML permettant l'ajout d'annotations indiquant si une définition est présente sur le serveur, le client, ou les deux.

Cette thèse présente la conception, la formalisation et l'implémention du langage ELIOM.

Remerciements

Ceux qui me connaissent savent que je ne suis pas particulièrement expansif, je serai donc bref: je remercie ma famille, mes amis, mes collègues et l'univers.

Remerciements, Deuxième essai

Je voudrais tout d'abord remercier mes deux directeurs de thèse, Roberto et Jérôme, qui ont formé une paire remarquablement complémentaire et m'ont beaucoup aidé dans tous les aspects de ma recherche, scientifiques ou non.

Je remercie l'équipe Ocsigen et les membres de l'Irill pour leur compagnie, conseils et pour avoir pris le temps d'écouter mes idées farfelues sur les prochaines fonctionnalités révolutionnaires qui vont faire d'Eliom le langage de programmation Web le plus mieux au monde!

Je remercie Jacques, pour cette petite remarque pendant ICFP 2015: "Tu devrais parler avec Oleg, ce que tu fais ressemble à MetaOCaml". Merci également à Oleg, pour ses remarques et ses encouragements.

Avant de commencer cette thèse, j'ai eu la chance de faire de nombreux stages. Je voudrais remercier tous les gens avec qui j'ai pu interagir à Édimbourg, Gotheborg et Grenoble. Ces stages m'ont beaucoup appris et surtout, je m'y suis énormément amusé! Merci également aux membres d'OCamllabs qui m'ont donné l'occasion de m'échapper de ma thèse pendant 3 mois pour faire joujou avec Mirage et boire du cidre dans les pubs de Cambridge.

En parlant de cidre, merci à tous les potes de promo de Rennes: Benoit, Simon, Nicolas, Clément, Gael et tous les autres. Merci également à David pour son encadrement pendant mes études et pour m'avoir orienté vers d'excellents stages.

Merci aux membres de l'IRIF avec qui j'ai pu interagir, notamment Vincent, Ralf, Jean-Baptiste et Michele avec qui j'ai eu le plaisir de faire mes enseignements et Alexis et Delia, pour leurs discussions et coups de pouce occasionnels. Je remercie particulièrement Odile, sans qui, soyons honnête, je n'aurais jamais réussi à faire quoi que ce soit. Merci à tous les doctorants de PPS, en particulier la folle équipe du bureau 3026: Clément, Yann, Rémi, Pierre, Amina, Charles, Léo, mon quotidien aurait été nettement plus terne sans vous.

Enfin, je remercie ma famille, pour à peu près tout.

Contents

1 Introduction

At the beginning, there was nothing. Suddenly, a burst of light, information and heated discussions: the Internet was born. And for a time, it was enough: Plain text information was exchanged between researchers, computers had 10Mb of RAM and there was no time for frivolous graphical Web pages. In 1989, Tim Berners-Lee disagreed and created the Encyclopedia Of Cats. To make his masterpiece accessible to everyone, he also created the World Wide Web: each page of the encyclopedia was accessed using the HyperText Transfer Protocol (HTTP). Pages were written using the HyperText Markup Language (HTML) and viewed through a Web browser. Related Web pages could be accessed by using Hyperlinks. Web pages could be easily published on the Internet using a Web server. This initiative was so successful that the World Wide Web now counts several billions of Web pages and consolidated cats as the dominant species on Earth.

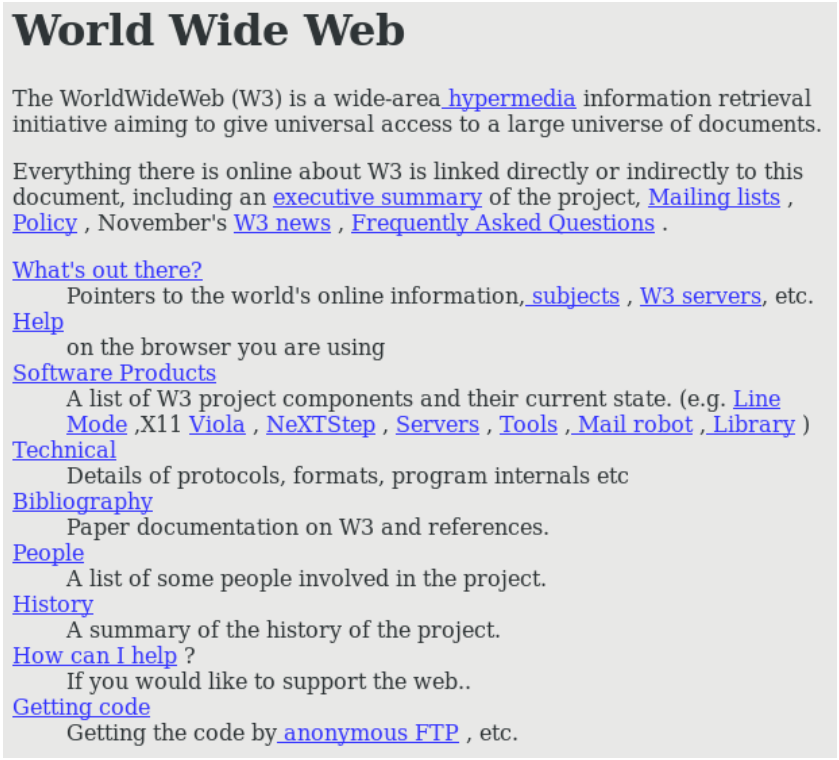


Figure 1.1: Tim Berners-Lee's first website (`http://info.cern.ch/`) as seen on a modern Web browser. A line-browser simulator is also available, for a more vintage experience.

Early websites such as Tim Berners-Lee's encyclopedia were static: the Web pages were written by hand and didn't contain any form of dynamic content. Quickly, people wrote programs to generate Web pages dynamically. In this case, pages are created when the user tries to access them. The programming of Web pages, or Web programming, has been done using a wide array of methods: CGI scripts, dedicated languages such as PHP, but also more generic languages using Web libraries such as Java, C# or Ruby. In all these cases, the idea is the same: a program written in the language in question receives a request from a client, usually a Web browser, and answers it by generating an HTML Web page. The Web page is then sent back to the client. This form of server-side dynamic website was embraced by early dedicated Web programming languages such as PHP (1994) and led to the creation of many popular "Web applications" such as Web search engines (1993), Web forums (around 1994[1]), Wikis (1995), etc. Many modern popular websites still mostly rely on this form of Web programming, for example Wikipedia and Google search.

An important limitation of these websites is that their dynamic behavior is limited to the server. Once the Web page is generated and sent to the client, it doesn't change until the client requested a new page. To obtain a more dynamic behavior, it was necessary to run programs inside the user's Web browser. Around 1995, multiple solutions for client-side scripting were developed: Java applets, Flash, and JavaScript. Numerous websites took advantage of these new capabilities to create interactive experiences available directly in the browser. In particular, Flash and Java allowed the developments of many browser-based games but also more serious applications such as interactive Web maps, music streaming websites, etc.

The combination of both server-side dynamic Web page generation and client-side scripting expanded the capability of websites and allowed to create rich Web applications that rival traditional desktop applications. This paved the way for many modern Web sites such as email clients, browser-based chats, collaborative text editors and more complex websites such as Facebook.

Unfortunately, this situation didn't come without some complexity. Consider the composition of an email in a Web client such as Gmail: While you type your message, a JavaScript program provides a rich text editor with bold, italic and emojis. The rich text is rendered in your browser using HTML for the content and CSS for the appearance. When you hit the send button, a dynamic AJAX request, which does not load a new page Web, sends a message containing your email to Google servers. On these servers, a Java program will store the content of the email in a database and send it to the desired recipient. This single task used two programming languages (JavaScript and Java), at least two description languages (HTML and CSS), all of it spread over two code bases and several distinct communication protocols. While Web applications grew more and more complex, with some subtle interplay of client and server behaviors and very large website with millions of line of code, the way we write Web application stayed similar: a client program, usually called "the frontend", handles the interactive part and communicates, usually through loosely structured text messages, to a server

[1] Although grumpy old-timer will say that Usenet and BBS did it all before.

program called "the backend". These two programs usually do not share code, not even the definition of the messages communicated, and are usually written by different teams in different programming languages, despite the fact that features used by users usually span client and server side indiscriminately.

1.1 On Web programming languages

The core of this issue is, of course, programming languages themselves. While the design of early Web programming languages might have been adapted to how the Web was used at the time, this is not the case anymore. We shall now explore some of the issues prevalent with existing Web languages.

1.1.1 The client-server dichotomy

Websites are usually split in two parts, also called "tiers": a backend that runs on a Web server and a frontend that runs in a browser. Hence, Web programmers must deal with communications between the backend and the frontend. These communications are often static: the backend generates (dynamically or not) a Web page associated with its frontend code and sends it to the browser, which then shows and executes it. Already with this simple scheme, problems can arise: the backend and the frontend should agree about the type and the shape of the generated Web page. For example, if the JavaScript program tries to find the HTML element named "user", the backend must have generated it; if the client program is generated dynamically, it should be correct; and so on.

Communications between frontend and backend can also be more complex. Consider the email client example given above: the request to send an email from the client to the server is dynamic and does not generate a new Web page. Communications can be done arbitrarily between client and server through various protocols (AJAXs, Web sockets, Comet, ...). In all these cases, the frontend and the backend must agree on which kind of data is sent. If the backend sends potatoes, the frontend should not expect tomatoes. All these guarantees are difficult to provide statically in today's Web programming languages and are usually enforced manually by the programmer. This is both time-consuming and error-prone.

This separation of Web applications as separate client and server programs is also problematic from a modularity point of view. Code must now be organized according to where it should run, and not what it does. Indeed, client and server functions can not be mixed arbitrarily, they can not even be in the same file! While this poses significant constraints on how a Web application is organized, it also severely restricts encapsulation. Internal details regarding communications must be accessible to the other side of the application, which often involves exposing them as a protocol to the whole application thus preventing the construction of good abstraction boundaries around individual libraries.

1.1.2 Tierless programming languages

Tierless Web programming languages aim to solve these issues by allowing programmers to build dynamic Web pages in a composable way. They allow programmers 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. A tierless program is then 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. 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.

1.1.3 Functional programming

Functional programming languages place the *function* at the center of the art of programming. Such languages allow to manipulate functions as first class values. More concretely, they offer the programmer the ability to create and manipulate functions seamlessly, allowing functions to return or take as argument other functions and enabling functions to be created, composed, stored, copied, etc, in arbitrary manners. Functional programming languages also favor immutable values: values that can not change during the lifetime of the program. To provide new information, new values must be created. Functional programming and immutable data-structures have been proven to provide great benefits in term of expressivity, safety and the ability to create extremely powerful abstractions for a wide variety of use cases as described most elegantly in the seminal essay "Why Functional Programming Matters" by Hughes [1989].

Functional programming was first introduced with the Lisp language by McCarthy [1960]. Since then, numerous languages provided support for functional programming: Scheme, OCaml, Haskell, Scala, but also more recently Swift and even Java and C++. Notably, JavaScript, the language of choice for frontend Web programming, provides support for functional programming which was leveraged to great effect in a large number of libraries such as Immutable or Reactjs. In a client-server context, functional programming was also used as a tool to structure the control flow across client-server boundaries [Balat, 2013, Queinnec, 2003, 2004, 2000].

Of particular interest is the ML language. ML, short for Meta Language [Milner, 1978], aimed to provide a new statically-typed functional programming language tailored for writing theorem provers. This language turned out to be very powerful and useful for a lot more than theorem proving. It inspired numerous languages which now form the larger "ML family". Some of its direct descendants, such as SML and OCaml, even enjoy large usage today. The main idea common to the numerous members of the ML family is to provide functional, sometimes impure, programming constructs along with a very rich static type system with type inference.

1.1.4 Static type systems

Type systems are an essential component of programming languages. They allow to label each element of the language by its "type" in order to ensure that it is used properly. This can be done dynamically at runtime, statically during a preliminary typechecking phase at compilation, or a mix of both. The choice between dynamically and statically typed languages is a long standing and very heated debate among programmers and numerous essays on the topic can be found online. In the context of this thesis, we will mostly consider the use of statically typed languages as the programming tool of choice. One reason is simply a strong preference from the author of this thesis. More objectively, static typing has been used to great effect in numerous contexts both to prevent bugs, but also to guide and inform the design of programs. Static typing can be further improved through type inference, which allows the programmer to partially or completely omit types annotations. This is notably the case of the ML family of functional programming languages, which combines static typing with a very powerful type inference mechanism to provide both the safety of statically typed languages and the flexible programming style of the more dynamic approaches.

In the context of tierless programming languages, a type system can also provide location information such as "where this code should run" and communication information such as "what am I sending". Combined with a strong static typing discipline, this ensures that the correctness of communications and library usage is checked statically by the compiler. This can even be done in a fairly non-intrusive way by leveraging type inference, ensuring that client and server expressions can be nested without the need for too many annotations.

1.1.5 Modularity and encapsulation

Mixing tiers directly inside expressions provides a very fined-grained notion of composition. However, programming large-scale software and libraries also requires the capacity for modularity and encapsulation at larger scale.

Modularity is a property ensuring that software can be constructed much like LEGO: by using independent bricks together to build a bigger structure. Each brick, or module, can use other modules but should be independent and self-contained from other non-related modules. Such modularity is also highly desirable in a tierless Web programming contexts. Indeed, parts of a library could be entirely on the server or on the client and programmers should be able to manipulate them freely. Unfortunately, most tierless languages do not support such modular approach to program architecture. Even in traditional Web programming languages such as JavaScript, support for proper modular programming is very recent (JavaScript modules were introduced in ES6, in 2015) and does not allow manipulating modules directly nor provides proper encapsulation.

Encapsulation is the complementary aspect of modularity: it allows to hide internal details of a library. Encapsulation improves the flexibility of a complex application: by hiding how each part works internally, the programmer should be able to swap them around as long as their external capabilities are the same. Encapsulation also improves

safety by ensuring that programmers can not misuse the internal details of a library by mistake. Encapsulation is essential for tierless Web programming: the details of how the server and client parts of a widget communicate with each other should not matter, as long as it provides the intended functionality. Communications should only be an internal implementation detail.

To solve these problems, we propose to leverage a well-known tool: ML-style modules.

ML modules On top of providing powerful functional programming constructs and static typing, modern ML languages feature a very expressive module system. In these languages, the module language is separate from the expression language. While the language of expression allows to program "in the small", the module language allows to program "in the large". In most languages, modules are compilation units: a simple collection of type and value declarations in a file. The SML module language [MacQueen, 1984] uses this notion of collection of declarations (called *structure*) and extends it with types (module specifications, or *signatures*), functions (parametrized modules, or *functors*) and function application, forming a small typed functional language.

In the history of ML languages, ML-style modules have been informally shown to be very expressive tools to architect software. Functors, in particular, allow to write generic implementations by abstracting over a complete module. Furthermore, ML modules provide very good encapsulation through the use of module signatures and abstract datatypes [Leroy, 1995, Crary, 2017]. Module signatures allow to restrict which members of a module are exposed to the outer world while abstract datatypes allow to hide the definition of a given type. Together, they allow to finely control the external interface of a library in order to enforce numerous properties. Another very desirable feature, called separate or incremental compilation, is the ability of the compiler to handle each module separately without needing to recompile all the dependent modules. This allows only the minimal amount of modules to be recompiled after a change which is essential for a fast development cycle. Separate compilation also improves modularity: indeed, checking that a module can indeed be compiled in isolation ensures that compilation can not fail later, in the context of a bigger program. Rich module systems such as ML's have been shown to provide excellent support for separate compilation [Leroy, 1994, Swasey et al., 2006].

Modular and tierless Associating a module language supporting proper modularity, encapsulation and separate compilation in a tierless programming setting is, however, quite delicate. Indeed, while separate compilation is available for languages with rich module systems such as ML, most tierless programming languages rely on whole program compilation to do the slicing. Furthermore, functors are very expressive and raise numerous issues when combined with tierless features that allows to interleave client and server sections.

1.2 Eliom

The Eliom language proposes to solve both fine-grained and large-scale modularity issues. Eliom is an extension of OCaml, an industrial-strength programming language of the ML family, for tierless web programming. Through the use of new syntactic constructs, it allows to create complete web applications as a single Eliom program describing both the client and the server behaviors. These new constructions allow statically-checked client-server communications, along with the ability to express rich client-server behaviors in a modular way. Although this increased expressivity could induce significant performance penalties, this is not the case thanks to an efficient compilation and execution scheme. Eliom programs are compiled statically in two parts: the first part runs on the server while the second part is compiled to JavaScript and runs on the client. Eliom also inherits the powerful type system and module system from OCaml, along with a rich ecosystem. This allows us to take advantage of numerous OCaml libraries, such as the rest of the Ocsigen project.

1.2.1 The Ocsigen project

Eliom is part of the larger Ocsigen project [Balat et al., 2009]. Ocsigen provides a comprehensive set of tools and libraries for developing Web applications in OCaml, including the compiler js_of_ocaml [Vouillon and Balat, 2014], a Web server, and libraries for concurrency [Vouillon, 2008], HTML manipulation [TyXML] and database interaction [Scherer and Vouillon, 2010]. Ocsigen libraries take deep advantage of the OCaml type system to provide guarantees about various aspects of client- and server-side Web programming. For example, HTML validity is statically guaranteed by the type system [TyXML]. These guarantees are complementary to the ones that Eliom provides.

Although this thesis will focus on the language extension, Eliom also comes with a large set of libraries for client and/or server programming that leverages the powerful new constructs introduced by the language. This notably includes RPCs; a functional reactive library for Web programming; a GUI toolkit [Ocsigen Toolkit]; a powerful session mechanism and an advanced service identification mechanism [Balat, 2014]. Some examples of these libraries, and how to implement them, are given in Chapter 2.

1.2.2 Principles of the Eliom language

All of the modules and libraries in Ocsigen, and in particular in the Eliom framework, are implemented on top of a unique core language. The design of this core language is guided by a set of six properties.

Explicit communications. Eliom uses manual annotations to 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. ELIOM relies on a novel and efficient execution model for client-server communication that avoids back-and-forth communication. This model is simple and predictable. Having a predictable execution model is essential in the context of an impure language, such as OCAML.

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

Integration with the host language. ELIOM is an extension of OCAML. 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, is a valid ELIOM module. Furthermore, we can specify if we want to use a given library on the client, on the server, or everywhere.

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. ELIOM leverages module abstraction to provide encapsulation and separation of concern for widgets and libraries. By combining module abstraction and tierless features, library authors can provide good APIs that do not expose the fine-grained details of client-server communication to the users.

Composition. The ELIOM language allows to define and manipulate *on the server*, as first class values, fragments of code which will be executed *on the client*. This gives us the ability to build reusable widgets that capture both the server and the client behaviors transparently. This makes it possible to define client-server building blocks (and libraries thereof) without further explicit support from the language.

This thesis will explore the consequences of these properties by presenting the design, the formalization and the implementation of the ELIOM language.

1.3 Plan

ELIOM aims to be a usable programming language. As such, we introduce ELIOM from a programming perspective in Chapter 2. Tierless programming goes further than just

gluing client and server pieces of code together. Through the various examples, we show that Eliom enables new programming idioms that allows to build web applications and libraries safely and easily while providing very good encapsulation and modularity properties.

As an extension of OCaml, Eliom is a compiled language. As for all compiled languages, there is a certain tension between the "intuitive" semantics, explained in term of syntactic reduction over the source language, which is easy to grasp by beginners, and the "real" semantics, which is expressed in term of compilation to a target language and the semantics of this target language. While the interpreted semantics is easier to explain, the implementation of Eliom uses the compiled semantics. The goal of the formalization of Eliom was precisely to resolve that tension by ensuring that both semantics agree. For this purpose, we proceed in several steps.

Eliom is not a language created in isolation, it extends the OCaml programming language. Similarly, our formalization of Eliom is based on a simpler ML language. The ML family of programming language is quite large and contains many different flavors. Our first step is to describe a simple ML language with modules which forms a reasonable subset of the OCaml language. This is done in Chapter 3.

In Chapter 4, we define the Eliom$_\varepsilon$ language, a subset of Eliom which is amenable to formalization, and describe its type system and interpreted semantics, along with various properties with relation to the base ML language. Eliom is compiled by producing two regular OCaml programs for each Eliom program. Similar, our compilation scheme for Eliom$_\varepsilon$ emits two ML programs with additional primitives. The compilation scheme and the target languages are described in Chapter 5, along with the simulation theorem which ensure that the interpreted semantics and the compiled semantics correspond.

This theorem, however, is not the end of the story. The implementation of programming languages rarely fits directly to the idealized formalization. This certainly applies to OCaml, and thus also to Eliom. In Chapter 6, we present the implementation of Eliom as an extension of OCaml. We expose its challenges and the choices we made to make the implementation possible. One particularly important property is the integration between Eliom and the vanilla OCaml compiler.

Finally, Chapter 7 explores the competitors and inspirations of the Eliom language, both in the field of tierless Web programming but also in various other domains such as distributed programming and staged meta-programming. Chapter 8 concludes with various remarks on tierless Web programming.

1.4 Contributions

Eliom, and the Ocsigen project, were initiated by Vincent Balat and Jérôme Vouillon several years ago. It enjoyed the input of many contributors along the years, who participated to the improvement and the refinement of various ideas you will find in this

thesis. In particular, fragments and sections and their compilation were already present in Eliom before this thesis started.

My contribution can be summarized as follows:

- A new type system for the Eliom constructs as an extension of the OCaml type system, including the notion of converters. Previous attempts at typing were limited at best, and unsound and hackish at worst.
- A new module system which fits the programming model of Eliom while preserving the good properties of the OCaml module system such as abstraction and modularity.
- A formalization of the type system, the interpreted semantics and the compilation scheme.
- A new implementation of the type checker, the compiler and the runtime of Eliom which more closely reflects the formalization.

More precisely, many examples and libraries presented in Chapter 2 have been developed along the years as part of the Ocsigen ecosystem, most recently by Vasilis Papavasilieou, Vincent Balat and Jérôme Vouillon. Furthermore, the compilation of sections, fragments and injections, along with the basic primitives that are presented in Chapter 5, are distilled versions of the original Eliom implementation due to Pierre Chambart, Grégoire Henry, Benedikt Becker and Vincent Balat.

2 Programming with Eliom

"A training course." I look at him. "What in? Windows NT system administration?"
He shakes his head. "Computational demonology for dummies."
Charles Stross, *The Atrocity Archive*

One of the goal of the Eliom language is to provide the essential building blocks for type-safe, efficient, tierless web programming. In order to demonstrate this, we first introduce Eliom's core concept in Section 2.1, then provide numerous examples. Our examples are extracted from code that appears in the Ocsigen tutorial [Tutorial] and in the Eliom library [Eliom]. Each example was chosen to illustrate a particular new programming pattern that is used pervasively in the Eliom ecosystem.

For clarity, we add some type annotations to make the meaning of the code clearer. These annotations are not necessary. While we use the OCaml language, we only assume some familiarity with functional languages.

2.1 Core concepts

Eliom only adds a few new constructions. The aim here is to provide constructions that are sufficiently minimal to be implemented on top of OCaml but also sufficiently expressive to provide convenient implementations of all the Web programming idioms as libraries.

2.1.1 Sections

In Eliom, we explicitly mark where a definition should be executed through the use of *section* annotations. We can specify whether a declaration is to be performed on the server or on the client as follows:

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

A third kind of section, written **shared**, is used for code executed on both sides. Sections allow the programmer to group related code in the same file, regardless of where it is executed.

In the rest of this thesis, we use the following color convention: client is in **yellow**, server is in **blue** and shared is in **green**. Colors are however not mandatory to understand the rest of this thesis.

2.1.2 Client fragments

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

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

For example, here, the expression `1 + 3` will be evaluated on the client, but it's possible to refer server-side to the future value of this expression (for example, put it in a list). The 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.

2.1.3 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:

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

The value inside the client fragment is extracted by `~%x`, whose value is `4` here.

Fragments and injections are not novel to this thesis. They draw inspiration from the very rich history of quotations for staging and macros [Bawden, 1999, Sheard and Jones, 2002] and have been used for several years in various tierless Web programming languages such as Hop [Serrano et al., 2006] and Eliom. These constructions are quite powerful and allow to express complex client-server interactions in convenient and safe ways, as we see in the rest of this section.

2.2 Client-server behaviors

The `hint_button` function below creates a button labeled "Show hint" that pops up a dialog box when activated. The message contained in the dialog box is a server-side

string that is given as argument to the function `hint_button`. One additional property is that the HTML is generated server-side and sent to the client as a regular HTML page.

For this purposes, we use an HTML DSL [TyXML] that provides combinators such as `button` and `a_onclick` (which respectively create an HTML tag and an HTML attribute). See Section 2.5.1 for more details on this DSL. The `~a` is the OCaml syntax for named arguments. Here, it is used for the list of HTML attributes.

Our function is implemented using a handler for 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 `msg` that contains the string to be showed to the user. The produced HTML fragment is shown in Example 2.1b and the inferred type in Example 2.1a. As we can see, this type does not expose the internal details of the widget's behavior. In particular, the communication between server and client does not leak in the API: This provides proper encapsulation for client-server behaviors. Furthermore, this widget is easily composable: the embedded client state cannot affect nor be affected by any other widget and can be used to build larger widgets.

```
let%server hint_button msg =
  button
    ~button_type:‘Button
    ~a:[a_onclick [%client fun _ -> alert ~%msg] ]
    [pcdata "Show hint"]
```

```
val%server hint_button : string -> Html.element
```

(a) Implementation and interface

```
<button onclick="...">
  Show hint
</button>
```

(b) Emitted HTML

Example 2.1: A button that shows a message

2.2.1 Introducing side effects

We now want to generalize our button widget by creating a button that increments a client-side counter and invokes a callback each time it is clicked. This is implemented by the `counter` function, shown below. This client function modifies the widget's state (the client-side reference `state`) and then calls the user-provided client-side callback `action`. This demonstrates that the higher-order nature of OCaml can be used in our client-server setting, and that it is useful for building server-side Web page fragments with parameterized client-side behaviors. In addition, note that the separation between `state` and `action` makes it straightforward to extend this example with a second button that decrements the counter while sharing the associated state.

```
let%server counter (action: (int -> unit) fragment) =
  let state = [%client ref 0 ] in
  button
    ~button_type:‘Button
    ~a:[a_onclick
          [%client fun _ -> incr ~%state; ~%action !(~%state) ]]
```

```
7     [pcdata "Increment"]
```

2.2.2 Communication and execution scheme

Our counter widget showcases complex patterns of interleaved client and server code, including passing client fragments as arguments to server functions, and subsequently to client code. This would be costly if the communication between the client and the server were done naively.

Eliom employs an efficient communication mechanism. Specifically, the server only ever sends data along with the initial version of the page. This is made possible by the fact that client fragments are not executed immediately when encountered inside server code. Intuitively, the semantics, presented formally in Section 4.3, is the following. When the server code is executed, the encountered client code is not executed right away; instead it is just registered for later execution once the Web page has been sent to the client. Only then is the client code executed. We also guarantee that client code, be it either client sections or fragments, is executed in the order that it was encountered on the server.

Let us consider the `counter` example above. When calling the `counter` function, we first encounter the `[%client ref 0]` fragment. We generate a fresh identifier that will be used to identify the result of the execution on the client. We store in a queue the fact that this piece of code should be run later. We also return the generated identifier. Later on, we encounter the fragment containing the callback called when the button is pressed: `[%client fun _ -> incr ~%state; ~%action !(~%state)]`. As before, we generate a fresh identifier and register this piece of code to be run on the client. We also send the content of injections `~%state` and `~%action`. Note that since `action` is a fragment, it is only represented by an identifier that will be used to find the value on the client. Once the server code has been executed, we send the information necessary for the client-side execution to the client. For example, that the identifier previously generated is associated to the result of `ref 0`. We can then execute the client-side part of the Eliom program. Using the sent information, we can execute client fragments and injections in the order expected by the programmer.

This presentation might make it seem as if we dynamically create the client code during execution of the server code. This is not the case. Like OCaml, Eliom is statically compiled and separates client and server code at compile time. During compilation, we statically extract the code included inside fragments and compile it as part of the client code to JavaScript. This allows us to provide both an efficient execution scheme that minimizes communication and preserve side effect orders while still presenting an easy-to-understand semantics. We also benefits from optimizations done by the `js_of_ocaml` compiler, thus producing efficient and compact JavaScript code.

2.3 Heterogeneous datatypes

Some datatypes are represented in fundamentally different ways on the server and on the client. This is a consequence of the different nature of the server and the client

environments. Eliom properly models this heterogeneous aspect by allowing to relate a client and a server datatype that share a similar semantics while having different definitions. We use this feature to present a safe and easy to use API for remote procedure calls (RPCs).

2.3.1 Remote procedure calls

When using fragments and injections, the only communication taking place between the client and the server is the original HTTP request and response. However, further communication is sometimes desirable. A remote procedure call is the action of calling, from the client, a function defined on the server. We present here an RPC API implemented using the Eliom language. The API is shown in Figure 2.1 and an example in Figure 2.2.

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

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

Figure 2.1: `Rpc` signature

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

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

Figure 2.2: Usage of the `Rpc` module

In Figure 2.2, we first create server-side an RPC endpoint using the function `Rpc.create`. Our example RPC adds 1 to its argument. The endpoint is therefore a value of type `(int,int)Rpc.t`, *i.e.*, an RPC whose argument and return values are both of type `int`. The type `Rpc.t` is abstract on the server, but is a synonym for a function type on the client. Of course, 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 in `~%plus1`, 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.

We can now combine the RPC API with the `counter` widget defined in Section 2.2.1 to create a button that saves the value of the counter on the server. This is presented in Example 2.2. We assume the existence of a `save_counter` function, which saves the counter in a database, and of the `counter` function defined previously. The signature of these functions are shown in Example 2.2a. We then proceed to define `save_counter_rpc` (*i.e.*, the server-side RPC interface for `save_counter`), and inject it into a fragment `f`. This fragment is subsequently used as the user-provided callback for `counter`. This way, each time the counter is incremented, its new value is saved server-side.

The RPC API we proposed 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, on the contrary, be made asynchronously: the client program keeps running while the call is made and the result is used when the communication is done. In the actual implementation, we use the Lwt library [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 thesis, we will simply 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 it is sent to the client via an injection is made possible by the use of *converters*. Figure 2.3 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 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 4.2.1.

We can use converters to implement the RPC API (Example 2.3). 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.

```
val%server save_counter :
  int -> unit
val%server counter :
  (int -> unit) fragment -> Html.t
```

(a) Environment

```
let%server save_counter_rpc : (int, unit) Rpc.t =
  Rpc.create save_counter

let%server widget_with_save : Html.element =
  let f = [%client ~%save_counter_rpc] in
  counter f
```

Example 2.2: Combination of the counter widget and the RPC API.

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

Figure 2.3: Schematized API for converters

2.3.3 Client-server reactive broadcasts

In the previous example, we used converters on rather simple datatypes: only a URL was sent, and a closure was created client-side. In this example, we use converters for a more ambitious API: lift *Functional Reactive Programming (FRP)* to be usable across client-server boundaries.

FRP is a paradigm that consists in operating on streams of data, either discrete (events) or continuous (signals). It has been used successfully to program graphical interfaces in a functional fashion, and can also be used to implement Web interfaces. Here, we show how to create an API that allows broadcasting server reactive events to a set of clients.

We assume pre-existing libraries implementing the following two APIs: An *untyped* broadcast API (Figure 2.4) and an FRP event API (Figure 2.5). Both of these APIs are orthogonal to Eliom's primitives; we can implement broadcast with standard Web techniques, and use the OCaml library React for FRP events. The broadcast API operates on messages of type `serial`, the serialization type introduced in Figure 2.3.

Let us now implement the typed broadcast API shown in Figure 2.6. It is quite similar to the RPC API: we have a type `t` with different implementations on the client and the server, and a server function `create` that takes a converter and an event stream as argument and produces a value of type `t`. Here, we use a converter explicitly in order to transfer elements on the broadcast bus.

The implementation of the `Broadcast` module is shown in Figure 2.7. On the server, a `BroadcastEvent.t` is composed of a converter that is used to transfer elements together with a URL. The `create` function starts by creating an untyped broadcast endpoint. We then use `Event.iter` to serialize and then send each occurrence of the provided event.

```
1  type%server ('i,'o) t = {
2    url : string ;
3    handler: 'i -> 'o ;
4  }
5
6  type%client ('i, 'o) t = 'i -> 'o
7
8  let%server serialize t = serialize_string t.url
9  let%client deserialize x =
10   let url = deserialize_string x in
11   fun i -> XmlHttpRequest.get url i
12
13 let conv = {
14   serialize = serialize ;
15   deserialize = [%client deserialize] ;
16 }
17
18 let%server create handler =
19   let url = "/rpc/" ^ generate_new_id () in
20   serve url handler ;
21   { url ; handler }
```

Example 2.3: Simplified RPC implementation corresponding to Figure 2.1.

We now need to create a converter for `BroadcastEvent.t`. We need to transmit two values: the URL of the broadcast endpoint, so that the client can subscribe, and the deserialization part of the provided converter, so that the client can decode the broadcasted messages. `raw_conv` provides a converter for a pair of a URL and a fragment. In addition to receiving this information, the client deserializer creates a new event stream and subscribes to the broadcast endpoint. We connect the broadcast output to the event stream by passing along all the (deserialized) messages.

As we can see in this example, we can use converters explicitly to setup very sophisticated communication schemes in a safe and typed manner. We also use the client deserialization step to execute stateful operations as needed on the client. Note that using a converter here allows effective use of resources: the only clients that subscribe to the broadcast are the ones that really need the event stream, since it has been injected.

2.4 Modules and signatures

In the previous examples, we declared simple Eliom modules containing client and server functions, along with signatures for such modules. We also used "pure" OCaml declarations that are neither client nor server such as the `serial` type (Figure 2.3) or the `Event` module (Figure 2.5). We now give a quick description of the OCaml module system and some of the Eliom extensions.

2.4.1 A primer on OCaml modules

The OCaml module system forms a second language separate from the expression language. While the language of expressions allows to program "in the small", the module

```
val%server t
val%server create : url -> t
val%server send : t -> serial -> unit

val%client subscribe :
  url -> (serial -> unit) -> unit
```

Figure 2.4: `Broadcast`: Untyped API

```
type 'a event
(** Events with occurrences of type ['a] *)

val create : unit -> 'a event * ('a -> unit)
(** [create ()] returns an event [e] and a
    [send] function *)

val iter : ('a -> unit) -> 'a event -> unit
(** [iter f e] applies [f] to
    [e]'s occurrences *)
```

Figure 2.5: `Event`: Reactive events API

```
type%server ('i, 'o) t
type%client ('i, 'o) t = 'o Event.event

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

Figure 2.6: `BroadcastEvent`: Shared reactive events API

language allows to program "in the large". In most languages, modules are compilation units: a simple collection of type and value declarations in a file. The ML module language uses this notion of collection of declarations (called *structure*) and extends it with types (module specifications, or *signatures*), functions (parametrized modules, or *functors*) and function applications, forming a small typed functional language.

In the previous examples, we already implicitely used the module system: each `.ml` 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` file. Let us now give a very simple example of functors. For a longer (and better) introduction to modules and functors, please consult the manual [Leroy et al., 2016] or the Real World OCaml

```
type%server ('i,'o) t = {
  conv : ('i, 'o) converter ;
  url : string ;
}
type%client ('i,'o) t = 'o Event.event

let%server create
    (conv : ('i, 'o) conv) (event : 'i Event.event) =
  let url = "/broadcast/" ^ generate_new_id () in
  let t = Broadcast.create url in
  let send x =
    Broadcast.send t (conv.serialize x)
  in
  let () = Event.iter send event in
  { conv ; url }

type%client ('i, 'o) t = 'o Event.event

let%server raw_conv
  : (url * 'a fragment, url * 'a) converter
  = Conv.pair Conv.url Conv.fragment

let%server serialize t =
  raw_conv.serialize (t.url, t.conv.deserialize)
let%client deserialize s =
  let url, deserial_msg =
    ~%raw_conv.deserialize s
  in
  let event, send = Event.create () in
  let handler msg = send (deserial_msg msg) in
  Broadcast.subscribe url handler ;
  event

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

Figure 2.7: `BroadcastEvent`: Shared reactive events. API shown in Figure 2.6.

book [Minsky et al., 2013].

Functors are simply functions that take a module and return another module. They can be used for a large variety of purposes. Here, we use them to build up data structure based on some primitive operations. Let us say we want to create dictionaries with keys of type `t`. One method to implement efficient dictionaries is to use Binary Search Trees, which requires a comparison function for values of type `t` in order to search in the tree. `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 2.9, we use this functor to create the `StringMap` module which defines dictionaries with string keys. We then define `d`, a dictionary which associates `"foo"` to `3`.

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

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

Figure 2.8: the `Map` module

```
module StringComp = struct
  type t = string
  let compare = String.compare
end
module StringMap = Map.Make(StringComp)

let d : int StringMap.t =
  StringMap.add "foo" 3 StringMap.empty
```

Figure 2.9: Dictionaries from strings to ints

2.4.2 Modules and locations

Section annotations are also available on module declarations, which allows to define client and server modules. One can also use regular OCaml modules and functors inside client and server code. For example, in Figure 2.10, we use `Map.Make` on the client to define maps whose keys are JavaScript strings. JavaScript strings are fairly different from OCaml strings, as they are represented by ropes instead of mutable byte arrays, hence the need for a different type. Note here that a functor from vanilla OCaml is applied to a client module and returns a client module.

As we saw in the previous examples, we can mix declarations from multiple locations inside the same module. Such modules are called "mixed". An important constraint is that, as you go down inside sub modules, locations should be properly included: A client module can not contain server declarations and conversely, but mixed modules can contain everything.

2.4.3 OCaml integration

We consider an additional location, "base", which can only contain OCaml constructs. Pure OCaml declarations such as `serial`, including pure OCaml modules, are considered of location base and are usable both on the client and on the server. Side-effecting

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

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

Figure 2.10: Map of js strings

base code will be executed on both the client and the server. Eliom guarantee that pieces of code inside base locations can only contain pure OCaml code, without any of the additional Eliom constructs. The semantics of base code is guaranteed to be exactly the same as the vanilla OCaml semantics. Furthermore, OCaml library can be imported and linked inside Eliom projects freely. It is even possible to use the compilation output of the regular OCaml compiler.

Additionally, it is possibly to decide that a given OCaml library should be imported in Eliom only client or server side. For example, we might want to use a vanilla OCaml database library in our Eliom project. We can simply specify that this library should only be loaded on the server and the Eliom type system will prevent its use it in client code.

2.4.4 Heterogeneous implementations

Shared sections make it possible to write code for the client and the server at the same time. This provides a convenient way of writing terse shared implementations, without duplicating logic and code. This does not necessarily entail that everything is shared. In particular, base primitives might differ between client and server, though the overall logic is the same. Just as we can implement heterogeneous datatypes with different client- and server-side representations, we can also provide interfaces common to the client and the server, with different client- and server-side implementations. We consider the case of database access. We first assume the existence of a server function `get_age` of type `string -> int` that performs a database query and returns the age of a person.

We can easily create a client version of that function via our RPC API of Figure 2.1.

```
let%server get_age_rpc = Rpc.create get_age
let%client get_age = %get_age_rpc
```

The API is then:

```
val%shared get_age : string -> int
```

We can use this function to write widgets that can be used either on the client or on the server:

```
let%shared person_widget name =
  div ~a:[a_class "person"] [
    text (name^" : "^string_of_int(get_age name))
  ]
```

This technique is used pervasively in Eliom to expose implementations than can be used either on the client or on the server with similar semantics, in a very concise way.

2.5 Mixed client-server data structures

We can readily embed client fragments inside server data structures. As a simple example of such a mixed data structure, consider a list of button names (standard server-side strings) and their corresponding client-side actions. Example 2.4 presents a function that takes such a list and builds an unordered HTML list of buttons.

Eliom makes such mixed data-structure particularly easy to write. Having explicit annotations with the usage of fragments is essential here. This would be quite difficult to achieve if the delimitation between client and server values were implicitly inferred.

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

Example 2.4: Function generating a list of buttons

2.5.1 HTML

A common idiom in Web programming is to generate the skeleton of a Web page on the server, then fill in the holes on the client with dynamic content, or bind dynamic client-side behaviors on HTML elements. In order to do that, the usual technique is to use the `id` or `class` HTML properties to identify elements, and to manually make sure that these identifiers are used in a coherent manner on the client and the server.

Eliom simplifies this process by mean of a client-server HTML library that allows injections of HTML elements to the client. Figure 2.11 shows a simplified API, which is uniform across clients and servers. The API provides combinators such as the `div` function shown below, which builds a `div` element with the provided attributes and child elements. We already used this HTML API in several previous examples.

On the server, HTML is implemented as a regular OCaml datatype. When sending the initial HTML document, this datatype is converted to a textual representation. This ensures compatibility with JavaScript-less clients and preserves the usual behavior of a Web server.

On the client, we represent HTML nodes directly as DOM trees. The mismatch between client and server implementations does not preclude us from providing a uniform API. However, to permit injections of HTML nodes from the server to the client, special care must be taken. In particular, we equip each injected node with an `id`, and `id` is the only piece of data sent by the serialization function. The deserialization function then

```
type%shared attribute
type%shared element

val%shared div :
  ?a:(attribute list) -> element list -> element

val%server a_onclick :
  (Event.t -> bool) fragment -> attribute

module%server Client : sig
  val node : element fragment -> element
end
```

Figure 2.11: `Html`: The simplified HTML API

finds the element with the appropriate `id` in the page. The `a_onclick` function finds the appropriate HTML element on the client and attaches the specified handler.

The fact that we use a uniform API allows us to abstract the specificities of the DOM and to provide other kinds of representations, such as a virtual DOM approach. A further improvement that fits in our design is nesting client HTML elements inside server HTML documents without any explicit DOM manipulation. This is done by the `Client.node` function (Figure 2.12), which takes a client fragment defining an HTML node and converts it to a server-side HTML node that can be embedded into the page. This function works by including a placeholder element server-side. The placeholder is later replaced by the actual element on the client.

```
let%server node (x: element fragment) : element =
  let placeholder = span [] in
  let _ = [%client
    let placeholder = ~%placeholder in
    let x = ~%x in
    Option.iter
      (Dom.parent placeholder)
      (fun parent ->
        Dom.replaceChild parent placeholder x)
  ] in
  placeholder
```

Figure 2.12: Implementation of `Client.node`

2.6 Shared values

We presented in Section 2.4.4 how we can use shared sections to write code that is used both on the client and on the server. Using the Eliom features we have described, we can also create *shared values*, which have a similar dual meaning, but at the level of expressions. The API is described in Figure 2.13 while the implementation is shown in Figure 2.14. Implementation of the converter for shared values is shown in Figure 2.15.

```
type%server ('a, 'b) shared_value =
  { srv : 'a ; cli : 'b fragment }
type%client ('a, 'b) shared_value = 'b

val%server local : ('a, 'b) shared_value -> 'a
val%client local : ('a, 'b) shared_value -> 'b

val%server cli :
  ('a, 'b) shared_value -> 'b fragment
val%client cli : ('a, 'b) shared_value -> 'b
```

Figure 2.13: Shared values API

```
let%server local x = x.srv
let%client local x = x

let%server cli x = x.cli
let%client cli x = x
```

Figure 2.14: Shared values Implementation

```
let%server shared_conv
  : (('a, 'b) shared_value, ('a, 'b) shared_value) converter
  = {
    serialize = (fun x -> Conv.fragment.serialize x.cli);
    deserialize = Conv.fragment.deserialize
  }
```

Figure 2.15: Converter for shared values

The server-side implementation of a shared value clearly needs to contain a fragment that can be injected on the client. On the other hand, the client cannot possibly inject a value on the server, so the client-side representation only consists of a fragment. For injecting a server-side shared value on the client, we use a converter whose server-side portion serializes only the fragment, and whose client-side portion deserializes this fragment.

2.6.1 Shared data-structures

Shared values are very useful when a given operation needs to be performed both on the server and on the client, but in a way that matches the specific requirement of each side. As an example, we present a *cached dictionary* API for storing data of interest on both the server and the client. This dictionary API should be well-adapted for Eliom's client-server style of programming. On the server, the dictionnary is to be used while serving a request, e.g., for locally caching data obtained from complex database queries. It is frequently the case that the client needs access to the same data; in that case, it is desirable that we avoid performing multiple RPCs. To achieve this, the semantics of the server-side addition operation (function `add`) is such that the value does not only become available for future server-side lookups, but also for client-side lookups. Of course, additional items may be added client-side, but then there is no expectation of server-side addition; the server-side dictionnary may not even exist any longer, given that it was local to the code handling the request.

As with the `Map` module, we want to define our dictionary over arbitrary keys. Let us first review what we need to define such a data-structure: We want two comparison function: one on the server and one on the client. We also need a converter that ensures

we can transmit values from the server to the client. For simplicity, we consider that the type of our keys is an OCaml base type. The type of our shared dictionaries `('a, 'b)table` contains two type variables `'a` and `'b`, corresponding to the server- and client-side contents respectively. The API in Figure 2.16a provides `add` and `find` operations, as is typical for association tables, which are available on both sides. The implementation is shown in Figure 2.16b. A dictionnary is implemented as a pair of a server-side dictionary and a client-side one. 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. Note that 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 `M` module on the client.

Several extensions of this API are possible. For pedagogic purposes, the type of key we used here is a pure OCaml type on the base location (Section 2.4.3). We could also have two different kinds of keys on the server and on the client, for example OCaml strings on the server and JavaScript strings on the client, which would ensure better efficiency. The function would then need two comparison functions and a converter between the two types. Alternatively, we could also easily create a full blown replicated dictionary: by using the RPC API, the client can require the server dictionary to be updated and by using the broadcast API, we can distribute new additions to all clients.

Going further, shared values empower an approach to reactive programming that is well-adapted for Eliom's client-server paradigm [Shared reactive programming]. This approach is the subject of ongoing work. One notable possible improvement is the notion of shared fragment, the analogous of shared declarations for expressions, which allow to avoid some code duplication present in the implementation in Figure 2.16b.

2.6.2 Mixed functors

In the previous example, we defined a functor which creates a module containing both client and server declarations. Such functors are called *mixed*. As we saw, mixed functors are fairly powerful and can be used for a wide variety of purposes. However, contrary to client and server functors, mixed functors are limited: arguments must be mixed modules, mixed structures and functors can not be nested arbitrarily and injections inside client-side bindings can only reference elements out of the functor. Injections inside client fragments can be used in arbitrary ways. A more precise description of these limitations is provided in Section 4.2.2.

2.7 A sophisticated example: accordions

We now demonstrate how to implement the well-known widget *accordion*. An accordion is a kind of application menu that displays collapsible sections in order to present information in a limited amount of space. The section titles are always visible. The content of a section is shown when the user clicks on its title. Only one section is open at a time.

```
module type T = sig
  type t
  val comparable : t -> t -> int
  val%server conv : (t, t) conv
end

module Cache (Key : T) : sig
  module M = Map.Make(Key)

  type%shared ('a, 'b) table =
    ('a M.t, 'b M.t) Shared.t

  val%shared add :
    Key.t -> ('a, 'b) Shared.t ->
    ('a, 'b) table -> ('a, 'b) table

  val%shared find :
    Key.t -> ('a, 'b) table ->
    ('a, 'b) Shared.t

  (* ... *)
end
```

(a) Signature

```
module Cache (Key : T) = struct
  module M = Map.Make(Key)

  type%shared ('a, 'b) table =
    ('a M.t, 'b M.t) Shared.t

  include%client M

  let%server add id v tbl =
    [%client M.add ~%id ~%v ~%tbl ];
    M.add id v.srv tbl.srv

  let%server find id tbl =
    { srv = M.find id tbl ;
      cli = [%client M.find ~%id ~%tbl]
    }

  (* ... *)
end
```

(b) Implementation

Figure 2.16: The `SharedTable` module

This widget can be used in numerous contexts such as the body of news articles, trip details in a train ticket search, etc.

In our example, sections are implemented independently and attached to the accordion given as parameter. The distinctive characteristic of our implementation, made possible by the two-level language, is that a section can be generated freely either on the server or on the client, and attached to an existing accordion. The example contains three sections, two generated server-side and one added dynamically client-side to the same accordion.

The code is shown in Figure 2.17. The data structure representing the accordion contains only a reference to a client-side function that closes the currently open section. Functions `new_accordion` and `accordion_section` are included in both the server and client programs (shared sections). Function `switch_visibility` is implemented client-side only. It just adds or removes an HTML class to the element, which has the effect of hiding or showing the element through CSS rules. Function `my_accordion` builds a server-side HTML page containing an accordion with two sections. It also sends to the client process, together with the page, the request to create the accordion (client fragment in function `new_accordion`) and to append a new section to the accordion. For this purpose, we use function `Client.node` on line 40.

```
let%client switch_visibility elt =
  if Class.contain elt "hidden"
  then Class.remove elt "hidden"
  else Class.add elt "hidden"

type%shared toggle =
  (unit -> unit) ref fragment

let%shared new_accordion () : toggle =
  [%client ref (fun () -> ()) ]

let%shared accordion_section
    (accordion : toggle) s1 s2 =
  let contents =
    div ~a:[a_class ["contents"; "hidden"]]
      [text s2]
  in
  let handler = [%client fun _ ->
    let toggle = ~%accordion in
    !toggle (); (*close previous section*)
    toggle :=
      (fun () -> switch_visibility ~%contents);
    switch_visibility ~%contents
  ]
  in
  let title =
    div
      ~a:[a_class ["title"]; a_onclick handler]
      [text s1]
  in
  div ~a:[a_class ["section"]]
    [title; contents]

let%server my_accordion () =
  let accordion = new_accordion () in
  div [
    accordion_section
      accordion
      "Item 1" "Server side generated" ;
    Client.node [%client
      accordion_section
        ~%accordion
        "Item 2" "Client side generated"
      ] ;
    accordion_section
      accordion
      "Item 3" "Server side generated" ;
  ]
```

Figure 2.17: The accordion widget

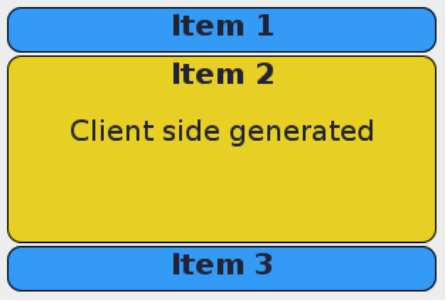


Figure 2.18: Resulting web page

2.8 Going further

Our examples demonstrate how the combination of fragments, injections and converters, along with the abstraction and modularity provided by the module system, can be used to build rich Web development libraries that provide convenient programming interfaces. While the module system and the notion of converter presented here are novel to this thesis; fragments, injections and abstraction have been leveraged in the Eliom library for many years to build very rich structures. In particular, the Eliom library provides uni- and bi-directional channels, progressive fetching of data, correct-by-construction links, and client-server reactive programming. Interestingly, a common pattern arising across these examples (just like for our RPC and HTML examples of Sections 2.3.1 and 2.5.1) is relating server and client datatypes that differ in their structure and APIs, but that have related intuitive meaning. Of course, the same building blocks and patterns can be used by the programmer to implement additional components outside the Eliom library, thus catering for their specific use cases.

3 The ML programming language

> *When I was a child, I was told that Santa Claus came in through the chimney, and that computers were programmed in binary code. Since then, I have learned that programming is better done in higher-level languages, more abstract and more expressive.*
>
> Xavier Leroy, *Polymorphic typing of an algorithmic language*

Programming languages of the ML family are like curry: everyone cook them differently and nobody agrees on the perfect ingredients, but they all end up being delicious. For ML, the original ingredients [Milner, 1978] are first class functions, parametric polymorphism, let bindings and algebraic datatypes. With the years, several new ingredients have been added to the mix; in particular mutable references, exceptions, pattern matching and a module system; which form the base of many languages in the ML family [Milner et al., 1990, Leroy et al., 2016].

Our version of ML 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 in the style of Leroy [1995]. We first present the syntax of the language in Section 3.1 and its type system in Section 3.2. We then present the semantics of this language in Section 3.3. Finally we indicate some relevant work in Section 3.4.

3.1 Syntax

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. Lists are noted with a star; for example τ^* is a list of type expressions. Indexed lists 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 3.1.

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

Expressions

$e ::= c$ (Constant)
$\mid x_i \mid p.x$ (Variables)
$\mid$ `Y` (Fixpoint)
$\mid (e\ e)$ (Application)
$\mid \lambda x.e$ (Function)
$\mid$ `let` $x = e$ `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)
$\mid \tau \to \tau$ (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)
$\mid$ `functor`$(X_i : \mathcal{M})M$ (Functor)
$\mid$ `struct` s `end` (Structure)

Structure body

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

Structure components

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

Programs

$P ::=$ `prog` S `end`

Module types

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

Signature body

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

Signature components

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

Environments

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

(b) The module language

Figure 3.1: ML grammar

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.

3.2 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.

3.2.1 The expression language

We introduce the following judgments:

$\Gamma \triangleright e : \tau$ The expression e has type τ in the environment Γ. See Figure 3.3.

$\Gamma \triangleright \tau_1 \approx \tau_2$ Types τ_1 and τ_2 are equivalent in environment Γ. See Figure 3.4.

$\Gamma \vDash \tau$ The type τ is well formed in the environment Γ. See Figure 3.2.

We note $\mathit{TypeOf}(c)$ 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 $\mathit{Close}(\Gamma, \tau)$. We use $(D) \in \Gamma$ to test if a given type or value is declared in the environment Γ. Note that for types, $(\texttt{type}\ (\alpha_i)\mathrm{t}) \in \Gamma$ 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:

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

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.

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

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[1] . This is done in rule ABSTYPEEQ.

$$\text{TYPEVAL}\ \dfrac{(\texttt{type}\ (\alpha_i)\text{t}) \in \Gamma \qquad \forall i,\ \Gamma \vDash \tau_i}{\Gamma \vDash (\tau_i)\text{t}} \qquad \text{ARROWVAL}\ \dfrac{\Gamma \vDash \tau_1 \qquad \Gamma \vDash \tau_2}{\Gamma \vDash \tau_1 \to \tau_2} \qquad \text{VARVAL}\ \dfrac{}{\Gamma \vDash \alpha}$$

Figure 3.2: Type validity rules – $\Gamma \vDash \tau$

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

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

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

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

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

Figure 3.3: ML expression typing rules – $\Gamma \triangleright e : \tau$

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

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

Figure 3.4: Type equivalence rules – $\Gamma \triangleright \tau \approx \tau'$

3.2.2 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 3.5.
$\Gamma \blacktriangleright \mathcal{M} <: \mathcal{M}'$	The module type $\mathcal{M}$ is a subtype of $\mathcal{M}'$ in Γ. See Figure 3.6.
$\Gamma \vDash \mathcal{M}$	The module type $\mathcal{M}$ is well-formed in Γ. See Figure 3.7.

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 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 2.4.1, 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 3.1. We assume the existence of two modules, presented in Example 3.1a. 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 3.1b. 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

```
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 3.1: Program using functors and manifest types

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.

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].

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 3.5 to 3.7 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 (val $a : \tau$). This is done in the QualModVar rule for the module language. The rules for the expression language are given in Figure 3.8.

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.

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

Strengthening The strengthening operation, noted $\mathcal{M}/p$, is defined in Figure 3.9 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.

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

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*[2]. 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:

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

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].

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

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

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 1.

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 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

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

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. □

3.2.3 Inference

Full inference is one of the greatest strength of the ML programming language. While we do not address inference formally in this thesis, here are some remarks. Inference is of course decidable for the core language using the well known $\mathcal{W}$ algorithm. It is "efficient", which means here that it is fast for usual programs, though pathological cases can be constructed. The typechecking rules for modules are not syntax directed. The Strength rule, in particular, is free floating. Leroy [1994] presents how to turn this into a syntax-directed type system, which allows inference as long as functor arguments are annotated.

3.3 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 Section 3.3.1.

We note v for values in the expression language and V for values in the module language. Values are defined in Figure 3.11. Values in the expression language can be either constants or lambdas. Module values are either structures, which are list of bindings of

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

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

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

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

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

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

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

Figure 3.5: Module typing rules – $\Gamma \blacktriangleright m : \mathcal{M}$

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

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

$$\frac{\Gamma \blacktriangleright \mathcal{M}'_a <: \mathcal{M}_a \quad \Gamma, (\texttt{module}\ X : \mathcal{M}'_a) \blacktriangleright \mathcal{M}_r <: \mathcal{M}'_r}{\Gamma \blacktriangleright \texttt{functor}(X : \mathcal{M}_a)\mathcal{M}_r <: \texttt{functor}(X : \mathcal{M}'_a)\mathcal{M}'_r}$$

$$\frac{\Gamma \triangleright \tau_1 \approx \tau_2}{\Gamma \blacktriangleright (\texttt{type}\ (\alpha^*)\mathrm{t_i} = \tau_1) <: (\texttt{type}\ (\alpha^*)\mathrm{t_i} = \tau_2)} \qquad \frac{}{\Gamma \blacktriangleright (\texttt{type}\ (\alpha^*)\mathrm{t_i}) <: (\texttt{type}\ (\alpha^*)\mathrm{t_i})}$$

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

Figure 3.6: Module subtyping rules – $\Gamma \blacktriangleright \mathcal{M} <: \mathcal{M}'$

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

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

$$\frac{\Gamma \vDash \tau \quad \mathrm{t_i} \notin \mathrm{BV}(\Gamma) \quad \Gamma; (\texttt{type}\ (\alpha^*)\mathrm{t_i} = \tau) \vDash \mathcal{S}}{\Gamma \vDash \texttt{type}\ (\alpha^*)\mathrm{t_i} = \tau;\ \mathcal{S}}$$

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

Figure 3.7: 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
$$\frac{\Gamma \blacktriangleright p : (\texttt{sig}\ \mathcal{S}_1; \texttt{type}\ (\alpha^*)\mathrm{t_i} = \tau; \mathcal{S}_2\ \texttt{end})}{\Gamma \triangleright (\tau_i)\mathrm{p.t} \approx \tau[n_i \mapsto p.n \mid n_i \in \mathrm{BV}(\mathcal{S}_1)][\alpha_i \mapsto \tau_i]_i}$$

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

Figure 3.8: Additional typing rules for the expression language

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

Figure 3.9: Module strengthening operation – $\mathcal{M}/p$

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 3.12. 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.

3.3.1 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 3.10. `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 `let` $x =$ (`print` 3) `in` (`print` $(x + 1)$)

$$\textsc{PrintTy}\quad \mathit{TypeOf}(\texttt{print}) = \forall\alpha.(\alpha \to \alpha) \qquad \textsc{PrintExec}\quad \delta(\texttt{print}, v) = v, \langle v \rangle$$

Figure 3.10: Typing and execution rules for `print`

Let us first show that e is of type `int`. The type derivation is provided in Example 3.2. 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 3.3. 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 (`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 $\langle 3; 4 \rangle$.

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

Example 3.2: Typing derivation for $e - \triangleright\, e : \texttt{int}$

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

Example 3.3: Execution derivation for $e - e \Rightarrow 4, \langle 3; 4 \rangle$

3.3.2 Modules

We now present an example of reduction involving modules. Our example program P is presented in Example 3.4a. 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 3.4b. 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.

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

(a) The program P

$$\dfrac{\dfrac{\dfrac{3 \Rightarrow 3, \langle\rangle}{\texttt{let } a = 3 \Rightarrow \{a{\mapsto}3\}, \langle\rangle}\,\textsc{ValDecl}}{\left(\begin{array}{l}\texttt{struct}\\ \quad\texttt{let } a = 3\\ \texttt{end}\end{array}\right) \Rightarrow \{a{\mapsto}3\}, \langle\rangle}\,\textsc{Struct} \qquad \dfrac{\dfrac{\dfrac{\rho(X) = V}{X \overset{\rho}{\Longrightarrow} V \equiv \{a{\mapsto}3\}, \langle\rangle} \qquad V(a) = 3}{X.a \overset{\{X{\mapsto}\{a{\mapsto}3\}\}}{\Longrightarrow} 3, \langle\rangle}\,\textsc{QualModVar}}{\texttt{let return} = X.a \overset{\{X{\mapsto}\{a{\mapsto}3\}\}}{\Longrightarrow} \{\texttt{return}{\mapsto}3\}, \langle\rangle}\,\textsc{ValDecl}\,\textsc{ModVar}$$

$$\dfrac{\left(\begin{array}{l}\texttt{module } X = \texttt{struct let } a = 3 \texttt{ end}\\ \texttt{let return} = X.a\end{array}\right) \Rightarrow \{X{\mapsto}\{a{\mapsto}3\}\} + \{\texttt{return}{\mapsto}3\}, \langle\rangle}{P \Rightarrow 3, \langle\rangle}\,\textsc{Program}\,\textsc{ModuleDecl}\,\textsc{ModuleDecl}$$

(b) Execution of P

Example 3.4: Example of execution with modules

Why big steps? One might wonder why use a big step semantics with traces, instead of a small step semantics. Indeed, small step usually make proofs easier, especially for simulations which we will use on ELIOM later. A first remark is that modules are not stable by substitution since (`struct ... end`).x is not valid (and is problematic to type). Hence we need to use a semantics with environments and closures. Let us now consider doing one step deep inside a structure using a small step semantics. The previously evaluated declarations in the structure should be available in the local environment which mean we would need to rebuild environments as we explore a context to execute a small step. We would also need to manipulate partially evaluated structures as we execute declarations. Furthermore, typing preservation for small steps would be difficult to express in the presence of abstract types. While this is all possible, big steps semantics with environments is, by comparison, fairly straightforward.

Expressions		**Modules**		**Bindings**	
$v ::= c$	(Constant)	$V ::= \{\ V_b^*\ \}$	(Structure)	$V_b ::= \{x_i \mapsto v\}$	(Values)
$\mid \lambda x.\rho.e$	(Function)	$\mid \texttt{functor}(\rho)(X_i : \mathcal{M})M$		$\mid \{X_i \mapsto V\}$	(Modules)

Figure 3.11: ML values

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

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

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

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

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

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

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

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

Figure 3.12: Big step semantics – $e \overset{\rho}{\Longrightarrow} v, \theta$

3.3.3 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 in this thesis. Instead, we give pointers to various relevant work containing such proofs.

Soundness for a small step semantics of our expression language is provided in 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, Tofte, 1988, Owens et al., 2016, Lee et al., 2007, Garrigue, 2009].

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. Notes that our language does not contain side-effects, which means that our language has a chance to be sound. The same cannot be said about OCaml in general.

3.4 Related works

Our formalization of the expression language is inspired by Wright and Felleisen [1994] which contains a small step semantics for the core language and extensions for references and exceptions, along with their soundness proofs. A discussion around the various styles of semantics in the context of traces is also provided by Nakata and Uustalu [2009]. The big step semantics takes inspiration from Owens et al. [2016] and Amin and Rompf [2017]. Tofte [1988] also presents a big-step semantics for ML and proves the soundness in the context of polymorphic type inference. A gentle introduction to soundness for big step semantics can be found in Siek [2013].

The landscape of ML modules is extremely rich, we only point to several key work that are directly relevant. Our module language is almost directly taken from Leroy [1994, 1995] which is the basis of the OCaml module language [Leroy et al., 2016]. The SML module system employs a different mechanism for propagating type equalities based on a `sharing` annotation [Milner et al., 1990]. Such annotation does not lend itself immediately to separate compilation [Swasey et al., 2006]. However, the SML module system has been (partially) mechanized [Lee et al., 2007]. A thorough and very instructive comparison between the various modules systems was done by Dreyer [2005, Chapter 1 and 2].

The module system we presented here is expressed directly in term of the syntax of the language. This is on purpose for two reasons. First, this is how it is implemented

in OCaml, making it possible to reason both about the formal system and the eventual implementation. Second, it makes the various extensions of Eliom fairly easy to model. In particular, we shall see that our compilation scheme for Eliom preserves the typing relation. Such results would be more delicate to express if the module system were expressed by elaboration. However, if such constraints were not considered, Rossberg et al. [2014] define an ML module system, including first class modules and applicative functors, in term of System F_ω. This yields a simple yet feature-full system that could prove easier to extend.

4 The Eliom programming language

> *Write a paper promising salvation, make it a 'structured' something or a 'virtual' something, or 'abstract', 'distributed' or 'higher-order' or 'applicative' and you can almost be certain of having started a new cult.*
>
> Edsger W. Dijkstra, *My hopes of computing science*

We now present the formalization of Eliom, a high-order applicative language for distributed client-server programming with strong support for abstraction and structured programming through modules[1]. The goal of this formalization is not to capture the complete Eliom language, nor the larger Ocsigen framework. Instead, we simply aim to capture the specific additions to the OCaml language from a typing and execution perspective. As such, we propose a new tierless calculus: Eliom$_\varepsilon$. In the Eliom$_\varepsilon$ language, programs are series of bindings returning a single client value which symbolizes the web page showed to the user. In particular, we do not try to capture the non-terminating aspect of a complete web server, nor the back-and-forth interactions of a web browser making HTTP requests to a server.

This might seem like a very limited formalism. However, when a browser requests a web page, each request follows a similar pattern: an HTTP request is made, the server executes a specific handler to answer this HTTP request and to send a web page and a client program to the browser, this client program is then executed. The cycle starts again when the user clicks on another link. Each handler can then be considered as its own individual program. In the practical context of Eliom and Ocsigen, this little program is executed in the context of a larger program: the web server, which can contain some state and be non-terminating. Our formalization, however, focuses on modeling the typing, execution and compilation of each little program that are run in handlers[2]. Eliom$_\varepsilon$ nevertheless captures several difficult points of Eliom, namely the handling of distinct type universes, the transmission of values between server and client stages, the detailed semantics, its interaction with side effects, the module system in the presence of stages, and a compilation process that supports separate compilation. This should hopefully keep most readers entertained and give us a stable foothold before attempting an implementation. While Eliom is an extension of OCaml, Eliom$_\varepsilon$ is an extension of the simple ML calculus with modules presented in Chapter 3. We first present the Eliom$_\varepsilon$ language in Section 4.1 and its type system in Section 4.2. We then present the semantics in Section 4.3 before stating various useful properties in Section 4.4

[1] Donations to the Cult of the Oxygenated Camel can be made by joining the OCaml consortium.
[2] Or, said in another way, Eliom$_\varepsilon$ is the calculus of modular tierless CGI scripts.

4.1 Syntax

We now present ELIOM$_\varepsilon$, a core language for ELIOM. ELIOM$_\varepsilon$ is an extension of the minimal ML language presented in Chapter 3, with both an expression and a module language. To emphasis 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.

The syntax is presented in Figure 4.4. It extends the syntax presented in Figure 3.1.

4.1.1 Locations

Before describing the syntax of ELIOM$_\varepsilon$, let us introduce the notation of *locations*. The grammar of locations is given in Figure 4.1. A location is "a place where the code runs". 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 forth 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 4.2, means that a variable (either a value, a type or a module) 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, while mixed declarations are not usable in base code. $\varsigma <: \varsigma'$, defined in Figure 4.3, 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. Note that both relations are reflexive: For instance, it is always possible to use client declarations when you are on the client.

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

Figure 4.1: Grammar of locations – ℓ and ς

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

Figure 4.2: "can be used in" relations on locations – $\ell \succ \ell'$

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

Figure 4.3: "can contain" relation on locations – $\varsigma <: \varsigma'$

Expressions

$$
\begin{array}{rll}
e ::= & \ldots & \\
& \mid \{\{\ e\ \}\} & \text{(Fragment)} \\
& \mid f\%v & \text{(Injection)} \\
f ::= & p.x \mid x_i \mid c & \text{(Converter)}
\end{array}
$$

Type Expressions

$$
\begin{array}{rll}
\tau ::= & \ldots & \\
& \mid \alpha_\ell & \text{(Type variables)} \\
& \mid \{\tau\} & \text{(Fragment types)} \\
& \mid \tau \rightsquigarrow \tau & \text{(Converter types)}
\end{array}
$$

Module Expressions

$$
\begin{array}{rll}
m ::= & \ldots & \\
& \mid \texttt{functor}_m(X_i : \mathcal{M})M & \text{(Mixed functor)}
\end{array}
$$

Module types

$$
\begin{array}{rll}
M ::= & \ldots & \\
& \mid \texttt{functor}_m(X_i : \mathcal{M}_1)\mathcal{M}_2 & \text{(Mixed functor)}
\end{array}
$$

Structure components

$$
\begin{array}{rl}
d ::= & \texttt{let}_\ell\ x_i = e \\
& \mid \texttt{type}_\ell\ (\alpha^*_{\ell_j})\mathrm{t_i} = \tau \\
& \mid \texttt{module}_\varsigma\ X_i = M
\end{array}
$$

Signature components

$$
\begin{array}{rl}
D ::= & \texttt{val}_\ell\ x_i : \tau \\
& \mid \texttt{type}_\ell\ (\alpha^*_{\ell_j})\mathrm{t_i} = \tau \\
& \mid \texttt{type}_\ell\ (\alpha^*_{\ell_j})\mathrm{t_i} \\
& \mid \texttt{module}_\varsigma\ X_i : \mathcal{M}
\end{array}
$$

Figure 4.4: Eliom$_\varepsilon$'s grammar

4.1.2 Expression language

The expression language is extended with two new constructs: fragments and injections. 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 manipulated on the server. An *injection* $f\%v$ can be used on the client to access values defined on the server. An injection must make explicit use of a converter f that specifies how to send the value. In particular, this should involve a serialization step, executed on the server, followed by a deserialization step executed 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 sometimes distinguish injections *per se*, which occur outside of fragments, and escaped values, which occur inside fragments.

The syntax of types is also extended with two constructs. A *fragment type* $\{\tau\}$ 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.

4.1.3 Module language

The main change in the module language of Eliom$_\varepsilon$ 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. 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.

4.2 Type system

Judgments are now annotated with a location that specifies where the given code should be typechecked. Judgments on the expression language can accept any core location ℓ while module judgments accept mixed locations ς. We note $\mathit{TypeOf}_\ell(c)$ 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 both a client and a server value with both the same name.

4.2.1 Expressions

The new typing rules for expressions are presented in Figures 4.5 to 4.7. We introduce two typing rules for the new constructions. 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*. Rule INJECTION is for the communication from the server to the client and can only be applied on the client. If e is of type τ_s on the server and f is of type $\tau_s \rightsquigarrow \tau_c$ on the server, then $f\%e$ is of type τ_c on the client. Since no other typing rules involves client fragments, it is impossible to deconstruct them.

The last difference with usual ML rules are the visibility of variable. As described earlier, bindings in ELIOM_m are located. Since access across sides are explicit, we want to prevent the use of client variables on the server, for example. In rule VAR, to use on location ℓ the variable v which is bound on location ℓ', we check that $\ell' \succ \ell$, defined in Figure 4.2, which means that the definition on ℓ can be used in ℓ'. Base elements b are usable everywhere. Mixed elements m are usable in both client and server. Type variables are also annotated with a location and follow the same rules. Using type variables from the client on the server, for example, is disallowed.

The validity judgement on types presented in Figure 4.6 is extended to check that locations are respected both for type constructors and type variables. This judgement is used in type declaration, which are presented in the module system.

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

Common rules

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

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

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

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

Server rules

$$\textsc{Fragment}\ \frac{\Gamma \rhd_c e : \tau}{\Gamma \rhd_s \{\{\ e\ \}\} : \{\tau\}}$$

Client rules

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

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

Figure 4.5: Eliom$_\varepsilon$ expression typing rules – $\Gamma \rhd_\ell e : \tau$

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

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

Figure 4.6: Type validity rules – $\Gamma \vDash_\ell \tau$

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

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

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

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

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

Figure 4.7: Type equivalence rules – $\Gamma \rhd_\ell \tau \approx \tau'$

$\tau_s \rightsquigarrow \tau_c$ is composed of a server-side encoding function of type $\tau_\mathrm{s} \to \texttt{serial}$, and a client-side decoding function of type $\texttt{serial} \to \tau_\mathrm{c}$. We assume the existence of two built-in converters:

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

Type universes

It is important to note that there is no identity converter (of type $\forall \alpha.(\alpha \rightsquigarrow \alpha)$). Indeed the client and server type universes are distinct and we cannot translate arbitrary types from one to the other. Some types are only available on one side: database handles, system types, JavaScript API types. Some types, while available on both sides (because they are in base for example), are simply not transferable. For example, functions cannot be serialized in 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.

Mixed datatypes

In Sections 3.2.1 and 3.2.2, we saw that 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 4.1. 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 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 4.4. 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.

```
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 4.1: A module with an abstract datatype.
Example 4.1a does not exposes information about acceptable sides for `'a` and `'b`. In Example 4.1b, annotations specifying the side of type variables are exposed in the interface.

4.2.2 Modules

We now detail ELIOM's module system. In the expression language, location transitions can only happen in very specific constructions: fragments and injections. This allow us to keep most of the ML type system unchanged. This is not the case anymore for modules: we allow users to create base, client and server modules, but also mixed modules that can contain base, client and server declarations, including other modules. This means we need to track locations quite precisely.

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.

Base location and specialization

In Section 2.4, we presented an example where a base functor `Map.Make`, is applied to a client module to obtain a new client module. As `Map.Make` is a module provided by the

standard library of OCaml, it is defined on location b. In particular, its input signature has components on location b, thus it would seem a module whose components are on the client or the server should not be accepted. We would nevertheless like to create maps of elements that are only available on the client. To do so, we introduce a specialization operation, defined in Figure 4.8, 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 Figure 4.2. This means that we never have to specialize a server module on the client (or conversely). Specialization towards location b has no effect since only base modules are accessible on location base. Specialization towards the location 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.

Mixed Functors

Mixed functors are functors declared in a mixed scope. 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 2.6.2 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 4.2, 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 4.2b 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 would be no value available to inject in the declaration of `y'`.

The solution here is that specialization on mixed functors should only specialize the return type, not the argument.

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 that unique name for lookup on the client.

```
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 4.2: A mixed functor using base declaration polymorphically

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 4.3. 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 4.3b, 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 4.3: 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 4.4a is not allowed while the one in Example 4.4b is allowed. Formally, this is guaranteed by the MixedFunctor rule, where each injection is typechecked in the outer typing environment.

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 4.3.

```
1
2 module%mixed F
3   (A:sig val%server x : int end)
4 = struct
5   let%client y = ~%A.x + 2
6 end
7
```

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

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

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

Example 4.4: Mixed functor and injections

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 4.2.2, is noted $\lfloor M \rfloor_\varsigma$ and is defined in Figure 4.8.

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

Subtyping and equivalence of modules

Subtyping rules are given in Figure 4.11. 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.

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

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

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

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

$$\lfloor \texttt{type}_\ell\ (\alpha^*_{\ell_j})\mathrm{t_i} = \tau; \mathcal{S} \rfloor_\iota = \begin{cases} \texttt{type}_\iota\ (\alpha^*_{\ell_j})\mathrm{t_i} = \tau; \lfloor \mathcal{S} \rfloor_\iota & \text{when } \ell \succ \iota \\ \lfloor \mathcal{S} \rfloor_\iota & \text{otherwise} \end{cases}$$

$$\lfloor \texttt{type}_\ell\ (\alpha^*_{\ell_j})\mathrm{t_i}; \mathcal{S} \rfloor_\iota = \begin{cases} \texttt{type}_\iota\ (\alpha^*_{\ell_j})\mathrm{t_i}; \lfloor \mathcal{S} \rfloor_\iota & \text{when } \ell \succ \iota \\ \lfloor \mathcal{S} \rfloor_\iota & \text{otherwise} \end{cases}$$

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

Where ι is either c or s.

Figure 4.8: Module specialization operation – $\lfloor M \rfloor_\varsigma$

$$\varepsilon/p = \varepsilon$$

$$(\texttt{sig}\ \mathcal{S}\ \texttt{end})/p = \texttt{sig}\ \mathcal{S}/p\ \texttt{end}$$

$$(\texttt{module}_\varsigma\ X_i = \mathcal{M}; \mathcal{S})/p = \texttt{module}_\varsigma\ X_i = \mathcal{M}/p; \mathcal{S}/p$$

$$(\texttt{type}_\ell\ (\alpha^*_{\ell_j})\mathrm{t_i} = \tau; \mathcal{S})/p = \texttt{type}_\ell\ (\alpha^*_{\ell_j})\mathrm{t_i} = (\alpha^*)\mathrm{p.t}; \mathcal{S}/p$$

$$(\texttt{type}_\ell\ (\alpha^*_{\ell_j})\mathrm{t_i}; \mathcal{S})/p = \texttt{type}_\ell\ (\alpha^*_{\ell_j})\mathrm{t_i} = (\alpha^*)\mathrm{p.t}; \mathcal{S}/p$$

$$(\texttt{val}_\ell\ x_i : \tau; \mathcal{S})/p = \texttt{val}_\ell\ x_i : \tau; \mathcal{S}/p$$

$$(\texttt{functor}(X_i : \mathcal{M})\mathcal{M}')/p = \texttt{functor}(X_i : \mathcal{M})(\mathcal{M}'/p(X_i))$$

$$(\texttt{functor}_m(X_i : \mathcal{M})\mathcal{M}')/p = \texttt{functor}_m(X_i : \mathcal{M})(\mathcal{M}'/p(X_i))$$

Figure 4.9: Module strengthening operation – M/p

$$\textsc{ModVar}\quad \frac{(\texttt{module}_{\varsigma'}\ X_i : \mathcal{M}) \in \Gamma \qquad \varsigma' \succ \varsigma}{\Gamma \blacktriangleright_\varsigma X_i : \lfloor \mathcal{M} \rfloor_\varsigma}$$

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

$$\textsc{Strength}\quad \frac{\Gamma \blacktriangleright_\varsigma p : \mathcal{M}}{\Gamma \blacktriangleright_\varsigma p : \mathcal{M}/p}$$

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

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

$$\frac{\Gamma \vDash_\varsigma \mathcal{M} \qquad \Gamma \blacktriangleright_\varsigma M : \mathcal{M}}{\Gamma \blacktriangleright_\varsigma (M : \mathcal{M}) : \mathcal{M}}$$

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

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

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

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

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

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

Figure 4.10: Module typing rules – $\Gamma \blacktriangleright_\varsigma m : M$

SubStruct

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

$$\frac{\Gamma \triangleright_{\ell_2} \tau_1 \approx \tau_2 \qquad \varsigma <: (\ell_1 \succ \ell_2)}{\Gamma \blacktriangleright_\varsigma (\texttt{val}_{\ell_1}\ x_i : \tau_1) <: (\texttt{val}_{\ell_2}\ x_i : \tau_2)}$$

$$\frac{\Gamma \blacktriangleright_{\varsigma_2} \mathcal{M}_1 <: \mathcal{M}_2 \qquad \varsigma <: (\varsigma_1 \succ \varsigma_2)}{\Gamma \blacktriangleright_\varsigma (\texttt{module}_{\varsigma_1}\ X_i : \mathcal{M}_1) <: (\texttt{module}_{\varsigma_2}\ X_i = \mathcal{M}_2)}$$

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

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

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

$$\frac{\varsigma <: (\ell_1 \succ \ell_2)}{\Gamma \blacktriangleright_\varsigma (\texttt{type}_{\ell_1}\ (\alpha^*_{\ell_j})\mathrm{t_i}) <: (\texttt{type}_{\ell_2}\ (\alpha^*_{\ell_j})\mathrm{t_i})} \qquad \frac{\Gamma \triangleright_{\ell_2} (\alpha^*_{\ell_j})\mathrm{t_i} \approx \tau \qquad \varsigma <: (\ell_1 \succ \ell_2)}{\Gamma \blacktriangleright_\varsigma (\texttt{type}_{\ell_1}\ (\alpha^*_{\ell_j})\mathrm{t_i}) <: (\texttt{type}_{\ell_2}\ (\alpha^*_{\ell_j})\mathrm{t_i} = \tau)}$$

$$\frac{\varsigma <: (\ell_1 \succ \ell_2)}{\Gamma \blacktriangleright_\varsigma (\texttt{type}_{\ell_1}\ (\alpha^*_{\ell_j})\mathrm{t_i} = \tau_1) <: (\texttt{type}_{\ell_2}\ (\alpha^*_{\ell_j})\mathrm{t_i})}$$

Figure 4.11: Module subtyping rules – $\Gamma \blacktriangleright_\varsigma M <: M'$

4.3 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 Chapter 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 5.4. As presented in Chapter 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 $\mathbf{X}.\mathbf{r}$. 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 4.13, 4.16 and 4.17. 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 $e \overset{\rho \,|\, \gamma \to \gamma'}{\Longrightarrow}_c v, \theta$ 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

4.3.1 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 4.12. The new evaluation rules are presented in Figure 4.13. The construction `bind env f` binds the current accessible environment to $\mathbf{f}$ in the global environment γ. This is implemented by the BindEnv rule. `bind r` $= e$ `with f` computes e in the environment previous associated to $\mathbf{f}$. The results 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.

In the rest of this thesis, 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.

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

Figure 4.12: Grammar of client programs

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

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

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

CLIENTCODE
Inherit the rules from ML

Figure 4.13: Semantics for client generated programs – $e \xrightarrow{\rho \mid \gamma \to \gamma'}_c v, \theta$

4.3.2 Base, Client and Server declarations

We now consider the case of base, client and server declarations. The rules are presented in Figure 4.16. 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 4.15. 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 $\tau_s \to \texttt{serial}$ 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 4.14. 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 4.5. 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 4.5: Execution of a client declaration

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

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

Figure 4.14: Execution contexts for injections – $E[\cdot]$

$$
\begin{aligned}
&\downarrow c = c && \text{when } c \in \mathit{Const}_b \\
&\downarrow \mathbf{p} = \mathbf{p} \\
&\downarrow v = \bot && \text{otherwise}
\end{aligned}
$$

Figure 4.15: Injections of values – $\downarrow v$

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 4.6. If we were simply to simply pass fragments along, the `print` statement would be evaluated twice. Instead, we create a fresh identifier **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 4.6. 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 4.6: Execution of a fragment containing side-effects

Closures and fragments In the client program above, we also use a reference **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 4.7. 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` **f**, where **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 **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. $\text{FRAGS}(D_s)$ returns the fragments syntactically present in D_s. For each fragment, the local environment is bound to the associated reference.

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

Example 4.7: Execution of a fragment inside a closure

Server code inside client contexts

$$\textsc{Injection}\ \frac{(f^s\ e) \overset{\rho}{\Longrightarrow}_s v, \mu, \theta}{f\%e \overset{\rho}{\Longrightarrow}_{c/s} (f^c\ \downarrow v), \mu, \theta} \qquad \textsc{ClientContext}\ \frac{\forall i,\ e_i \overset{\rho}{\Longrightarrow}_{c/s} v_i, \mu_i, \theta_i}{E[e_1, \ldots, e_n] \overset{\rho}{\Longrightarrow}_{c/s} E[v_1, \ldots, v_n], \mu_1; \ldots; \mu_n, @_i\, \theta_i}$$

Server code

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

ServerCode
Inherit the rules from ML

Base code

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

Declarations

$$\textsc{BaseDecl}\ \frac{D_b \overset{\rho}{\Longrightarrow}_b V, \varepsilon, \theta \qquad S \overset{\rho+\text{V}}{\Longrightarrow}_m V', \mu', \theta'}{D_b; S \overset{\rho}{\Longrightarrow}_m V + V', (D_b; \mu'), \theta \mathbin{@} \theta'} \qquad \textsc{ClientDecl}\ \frac{D_c \overset{\rho}{\Longrightarrow}_{c/s} \overline{D_c}, \mu, \theta \qquad S \overset{\rho}{\Longrightarrow}_m V, \mu', \theta'}{D_c; S \overset{\rho}{\Longrightarrow}_m V, (\mu; \overline{D_c}; \mu'), \theta}$$

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

$$\textsc{Program}\ \frac{P \overset{\rho}{\Longrightarrow}_m (), \mu, \theta_s \qquad \mu \overset{\rho\,|\,\varepsilon \to \gamma}{\Longrightarrow}_c v, \theta_c}{P \overset{\rho}{\Longrightarrow} v, \theta_s \mathbin{@} \theta_c}$$

Figure 4.16: Semantics for base, client and server sections – $e \overset{\rho}{\Longrightarrow}_\varsigma v, \mu, \theta$

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 $\texttt{struct} \ldots \texttt{end}_{\mathbf{F}}$.

4.3.3 Mixed modules

Let us now describe the reduction relation for mixed modules. The mixed reduction relation is presented in Figure 4.17 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 4.8. 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 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.

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 4.9. The server code proceed in the following way: we first define a

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

Example 4.8: Execution of mixed modules

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 $\texttt{int}\%x$ 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. 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.

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

Example 4.9: Execution of mixed functors with injections

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 4.10. 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, $\texttt{module}\ X = \mathbf{Y}$. This gives us a complete client structure which we can bind to $\mathbf{R_Z}$.

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

Example 4.10: Execution of mixed functors with fragments

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 of this method is that it leads to two executions of the client side. We shall discuss this in Section 5.5.

Mixed module expressions

$$\textsc{MixedStruct}\quad \frac{S[\mathbf{f_i} \mapsto \mathbf{X}.\mathbf{f_i}]_i \overset{\rho}{\Longrightarrow}_m V, \mu, \theta \qquad V' = V + \{\texttt{Dyn} \mapsto \mathbf{X}\}}{\texttt{struct}\ S\ \texttt{end}_{\mathbf{X}} \overset{\rho}{\Longrightarrow}_m V', \mathbf{X}, \texttt{bind}\ \mathbf{X} = \texttt{struct}\ \mu\ \texttt{end}, \theta}$$

$$\textsc{MixedModVar}\quad \frac{\rho(X) = V}{X \overset{\rho}{\Longrightarrow}_m V, X, \varepsilon, \langle\rangle}$$

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

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

$$\textsc{NotStructBeta}\quad \frac{V = \texttt{functor}_m(\rho')(X_i : \mathcal{M}_i)_i M \qquad M \overset{\rho'+\{\mathrm{X_i}\mapsto\mathrm{V_i}\}_\mathrm{i}}{\Longrightarrow}_m V_r, \mu, \theta}{V(V_1)\ldots(V_n) \overset{\rho}{\Longrightarrow}_m V_r, \mu, \theta,}$$

$$\textsc{MixedQualModVar}\quad \frac{p \overset{\rho}{\Longrightarrow}_m V, \mu, \theta}{p.X \overset{\rho}{\Longrightarrow}_m V(X), p.X, \mu, \theta}$$

$$\textsc{Empty}\quad \frac{}{\varepsilon \overset{\rho}{\Longrightarrow}_m \{\}, \varepsilon, \langle\rangle}$$

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

Mixed declarations

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

Figure 4.17: Semantics for mixed modules – $M \overset{\rho}{\Longrightarrow}_\varsigma V, \mu, \theta$

4.4 Results on locations

The behavior of locations and specialization in the presence of the various language constructs is not obvious, even with various examples. We now present various elementary results related to locations that should make the behavior of some constructions easier to grasp and inform the design of our compilation scheme. Let us note $\mathcal{M}_{[\ell\mapsto\ell']}$ the substitution on locations in an Eliom module type $\mathcal{M}$.

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. □

4.4.1 Relation between ML and Eliom

Eliom is an extension of ML. However, we also want Eliom to be well integrated in its host language. We now detail the relation between ML and Eliom. In particular, we show that ML can be completely embedded in Eliom without changes.

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.

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

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

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 $\mathcal{M}$:

$$\Gamma \vDash_b \tau \implies \Gamma_{[\mapsto \mathrm{ML}]} \vDash_{\mathrm{ML}} \tau \qquad \Gamma \vDash_b \mathcal{M} \implies \Gamma_{[\mapsto \mathrm{ML}]} \vDash_{\mathrm{ML}} \mathcal{M}_{[\mapsto \mathrm{ML}]}$$
$$\Gamma \rhd_b e : \tau \implies \Gamma_{[\mapsto \mathrm{ML}]} \rhd_{\mathrm{ML}} e : \tau \qquad \Gamma \blacktriangleright_b M : \mathcal{M} \implies \Gamma_{[\mapsto \mathrm{ML}]} \blacktriangleright_{\mathrm{ML}} M_{[\mapsto \mathrm{ML}]} : \mathcal{M}_{[\mapsto \mathrm{ML}]}$$

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:

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

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

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

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 to 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.

4.4.2 Notes on soundness

We do not give a complete proof of soundness for Eliom$_\varepsilon$. One of the reason is that a soundness proof for our version of ML is already out of the scope of this thesis (See Section 3.3.3). The Eliom$_\varepsilon$ semantics is mostly implemented in terms of the ML one. Here we will assume that the ML semantics is sound, and give some arguments towards the soundness of Eliom$_\varepsilon$.

Server and Base semantics Since base code exactly corresponds to ML code, and can not depend on server and client identifiers, the soundness of the base reduction relation is equivalent to the soundness of ML. The server reduction relation only adds the Fragment rule. If a fragment is well typed and the various injection can be reduced, it is easy to see that this rule always applies. Since references $\mathbf{r}$ are opaque values on the server and can only be used in injections, they do not compromise the soundness of the server reduction relation.

Client emitted code In our presentation, we do not give typing rules for the `bind env` and the `bind with` constructs. Such typing rules can simply be given by typing $\mathbf{f_i}$ references similarly to environments: with a module type. With this addition, it should be possible to show that client emitted programs are always well typed. We can then rely on the soundness of ML equipped with `bind`. The difficult point here are fragments, due to the delayed nature of the `bind` constructs. However, the content of a fragment is always executed in an environment with a type compatible with its typing environment, which should ensure correctness.

Mixed modules The client-side behavior of a mixed module is fairly easy to model, since it is equivalent to a client module. The server-side behavior is mostly the same as the one of a pair composed by a server-side module and a fragment containing a client-side module. As such, the soundness arguments should be the same that the one used for fragments.

5 Compilation of Eliom programs

In Chapter 4, 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 Chapter 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 thesis (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. However, we also want to ensure that the interpreted semantics, which is explained to users, corresponds to the compiled semantics[1]. This is done in Section 5.4. Finally, we discuss the design of mixed functors from a compilation perspective in Section 5.5.

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

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 5.1. 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: $\mathbf{r}$. References can also be paths, such as $\mathbf{X}.\mathbf{r}$. In some contexts, we accept a special form of reference path, noted $\texttt{Dyn}.\mathbf{x}$ 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`.

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 $\tau_\mathrm{s} \to \texttt{serial}$ and f^c is of type $\texttt{serial} \to \tau_\mathrm{c}$. 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.

[1] Thus, the main result of this chapter is that you do not need to read it to understand ELIOM$_\varepsilon$ programs!

$\mathrm{ML_S}$ grammar

$\mathbf{p} ::= \texttt{Dyn.}\mathbf{f} \mid (\mathbf{X}.)^*\mathbf{f}$ (Reference path)
$\tau ::= \ldots \mid \texttt{frag}$ (Fragment type)
$v ::= \ldots \mid \mathbf{r}$ (Reference)
$e ::= \ldots$
$\mid \texttt{fragment}\ \mathbf{p}\ e^*$ (Fragment call)
$M ::= \ldots$
$\mid p.\texttt{Dyn}$ (Dynamic field)
$\mid \texttt{fragment}_m\ \mathbf{p}\ (M^*)$ (Fragment)
$D ::= \ldots$
$\mid \texttt{injection}\ \mathbf{x}\ e$ (Injection)
$\mid \texttt{end}\ ()$ (End token)

$\mathrm{ML_C}$ grammar

$\mathbf{p} ::= \texttt{Dyn.}\mathbf{f} \mid (\mathbf{X}.)^*\mathbf{f}$ (Reference path)
$e ::= \ldots \mid \mathbf{x}$ (Reference)
$M ::= \ldots \mid \mathbf{X}$ (Module reference)
$D ::= \ldots$
$\mid \texttt{bind}\ \mathbf{p} = e$ (Fragment closure)
$\mid \texttt{bind}_m\ \mathbf{p} = M$ (Functor fragment)
$\mid \texttt{exec}\ ()$ (Fragment execution)

Figure 5.1: Grammar for $\mathrm{ML_S}$ and $\mathrm{ML_C}$ as extensions of ML_ε

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 5.1. 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 ($\texttt{int} \rightsquigarrow \texttt{int}$) and the associated encoding and decoding functions. The server program, in Example 5.1a, 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 5.1b, 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 5.1c. After the execution of the server code, this mapping is sent to the client, and used to execute the client code.

```
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 5.1: Client-server programs calling and using injections

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 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` $\mathbf{p}$ $e_1 \dots e_n$ 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 5.2 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 5.2a) calls $\mathbf{f}$ with 2 as argument, injects the result and then calls $\mathbf{f}$ with 3 as argument. The client code, in Example 5.2b, 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 5.2c) and the mapping of injections (Example 5.2d) 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 4.3.1. 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

```
{r1↦(f 2)}; end;
{r2↦(f 3)}; end;
```

(c) List of fragments

$$\mathbf{v_1} \mapsto \mathbf{r_1}$$

(d) Mapping of injections

Example 5.2: 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 $\mathbf{p}$.
- $\texttt{fragment}_m$ $\mathbf{p}$ $(\mathbf{R_1}) \dots (\mathbf{R_n})$ 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.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 5.3.

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 4.3, we add a

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

`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 4.3.3, 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 $\mathbf{R_1.f}$ fragment closure. Since it is the result of this specific functor application, the closure $\mathbf{R_1.f}$ 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.

```
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

```
{R1↦F X0};
{r2↦R1.f 2};
end
```

(c) List of fragments

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

(d) Mapping of injections

Example 5.3: Client-server program using module fragments

5.1.5 Type system rules

The ML_S and ML_C typing rules are presented in Figures 5.2 and 5.3 as a small extension over the ML typing rules presented in Section 3.2. Note that the typing rules for the new primitives are weakly typed and are certainly not sound with respect to serialization and deserialization. Given arbitrary ML_S and 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. ML_S and 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. ML_S and 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 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 Section 3.3). The $\Rightarrow_{\text{ML}_\text{s}}$ reduction for ML$_\text{s}$ server programs is presented in Figure 5.4. The $\Rightarrow_{\text{ML}_\text{c}}$ reduction for ML$_\text{C}$ client programs is presented in Figure 5.5. Let us consider a server structure S_s and a client structure S_c. A paired execution of the two structures is presented below:

$$S_s \xRightarrow{\rho_\text{s}}_{\text{ML}_\text{s}} V_s, \xi, \zeta, \theta_s \qquad\qquad S_c, \xi \xRightarrow{\rho_\text{c} \mid \gamma \cup \zeta \rightarrow \gamma'}_{\text{ML}_\text{c}} V_c, \xi', \theta_c$$

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 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 4.15. This models the serialization

$$\text{FRAGMENT}\ \frac{\forall i,\ \Gamma \triangleright_{\text{ML}_\text{s}} e_i : \texttt{serial}}{\Gamma \triangleright_{\text{ML}_\text{s}} \texttt{fragment}\ \mathbf{p}\ e_1\ \dots\ e_n : \texttt{frag}} \qquad \text{INJECTION}\ \frac{\Gamma \triangleright_{\text{ML}_\text{s}} e : \texttt{serial}}{\Gamma \blacktriangleright_{\text{ML}_\text{s}} \texttt{injection}\ \mathbf{v}\ e : \varepsilon} \qquad \text{END}\ \frac{}{\Gamma \blacktriangleright_{\text{ML}_\text{s}} \texttt{end}\ () : \varepsilon}$$

$$\text{FRAGMENT}_m\ \frac{\forall i,\ \Gamma \triangleright_{\text{ML}_\text{s}} p_i : \mathcal{M}_\text{i}}{\Gamma \blacktriangleright_{\text{ML}_\text{s}} \texttt{module Dyn} = \texttt{fragment}_m\ \mathbf{p}\ p_1.\texttt{Dyn}\ \dots\ p_n.\texttt{Dyn} : \varepsilon}$$

Figure 5.2: Typing rules for ML$_\text{s}$

$$\text{BIND}\ \frac{\Gamma \triangleright_{\text{ML}_\text{c}} e : \tau}{\Gamma \blacktriangleright_{\text{ML}_\text{c}} \texttt{bind}\ \mathbf{p} = e : \varepsilon} \qquad \text{BIND}_m\ \frac{\Gamma \blacktriangleright_{\text{ML}_\text{c}} M : \mathcal{M}}{\Gamma \blacktriangleright_{\text{ML}_\text{c}} \texttt{bind}_m\ \mathbf{p} = M : \varepsilon} \qquad \text{REFERENCE}\ \frac{}{\Gamma \triangleright_{\text{ML}_\text{c}} \mathbf{x} : \texttt{serial}} \qquad \text{EXEC}\ \frac{}{\Gamma \blacktriangleright_{\text{ML}_\text{c}} \texttt{exec}\ () : \varepsilon}$$

Figure 5.3: Typing rules for ML$_\text{s}$

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.`$\mathbf{f}$, it must first resolves `Dyn` to the actual reference. This is done through the DYN rule.

Segmented execution In $\mathrm{ML_S}$ and $\mathrm{ML_C}$ 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 $\mathbf{r}$ 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.

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

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

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

$$\textsc{End}\quad \frac{S \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} V, \xi, \zeta, \theta}{\texttt{end}\ (); S \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} V, \texttt{end}{+\!\!+}\xi, \zeta, \theta} \qquad \textsc{Dyn}\quad \frac{\rho(\texttt{Dyn}) = \mathbf{R}}{\texttt{Dyn.f} \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} \mathbf{R}.\mathbf{r}}$$

$\textsc{ServerCode}$ Inherit the rules from ML

Figure 5.4: Semantics rules for $\mathrm{ML_s}$– $S \overset{\rho}{\Longrightarrow}_{\mathrm{ML_s}} V, \xi, \zeta, \theta$

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

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

$$\textsc{Reference}\quad \frac{\gamma(\mathbf{r}) = v}{\mathbf{r}, \xi \xrightarrow{\rho\,|\,\gamma \to \gamma}_{\mathrm{ML_c}} v, \xi, \langle\rangle} \qquad \textsc{Exec}\quad \frac{\xi \xrightarrow{\rho\,|\,\gamma \to \gamma'}_{\mathrm{ML_c}} \{\}, \theta \qquad S, \xi' \xrightarrow{\rho\,|\,\gamma' \to \gamma''}_{\mathrm{ML_c}} V, \xi'', \theta'}{(\texttt{exec}\ (); S), \xi{+\!\!+}\texttt{end}{+\!\!+}\xi' \xrightarrow{\rho\,|\,\gamma \to \gamma''}_{\mathrm{ML_c}} V, \xi'', \theta \mathbin{@} \theta'}$$

$$\textsc{Frag}\quad \frac{(\mathbf{f}\ v_1 \ldots v_n) \xrightarrow{\rho\,|\,\gamma \to \gamma'}_{\mathrm{ML_c}} v, \theta \qquad \xi \xrightarrow{\rho\,|\,\gamma' + \{\mathbf{r} \mapsto \mathrm{v}\} \to \gamma''}_{\mathrm{ML_c}} \{\}, \theta'}{\{\mathbf{r} \mapsto \mathbf{f}\ v_1 \ldots v_n\}{+\!\!+}\xi \xrightarrow{\rho\,|\,\gamma \to \gamma'}_{\mathrm{ML_c}} \{\}, \theta \mathbin{@} \theta'} \qquad \textsc{Dyn}\quad \frac{\rho(\texttt{Dyn}) = \mathbf{R}}{\texttt{Dyn.f} \overset{\rho}{\Longrightarrow}_{\mathrm{ML_c}} \mathbf{R}.\mathbf{r}}$$

$$\textsc{Frag}_m\quad \frac{\mathbf{F}(\mathbf{R}_1) \ldots (\mathbf{R}_n) \xrightarrow{\rho \cup \{\texttt{Dyn} \mapsto \mathbf{R}\}\,|\,\gamma \to \gamma'}_{\mathrm{ML_c}} V, \theta \qquad \xi \xrightarrow{\rho\,|\,\gamma' + \{\mathbf{R} \mapsto \mathrm{V}\} \to \gamma''}_{\mathrm{ML_c}} \{\}, \theta'}{\{\mathbf{R} \mapsto \mathbf{F}\ \mathbf{R}_1 \ldots \mathbf{R}_n\}{+\!\!+}\xi \xrightarrow{\rho\,|\,\gamma \to \gamma'}_{\mathrm{ML_c}} \{\}, \theta \mathbin{@} \theta'}$$

$\textsc{ClientCode}$ Inherit the ML rules

Figure 5.5: Semantics rules for $\mathrm{ML_c}$– $S, \xi \xrightarrow{\rho\,|\,\gamma \to \gamma'}_{\mathrm{ML_c}} V, \xi', \theta$

5.2 Compilation

In Section 5.1, we presented the two target languages ML_S and ML_C. We now present the compilation process transforming *one* ELIOM_ε program into two distinct ML_S and ML_C programs. Before giving a more formal description in Section 5.2.2, we present the compilation process through three examples of increasing complexity.

Injections and fragments Example 5.4 presents an ELIOM_ε program containing only simple declarations involving fragments and injections without modules. The ELIOM_ε program is presented on the left, while the compiled ML_S and ML_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 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 $\{\!\{ \texttt{frag}\%a + 1 \}\!\}$ 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_ε	ML_S	ML_C
$\texttt{let}_s\ a = \{\!\{ 1 \}\!\};$	$\texttt{let}\ a = \texttt{fragment}\ \mathbf{f_0}\ ();$	$\texttt{bind}\ \mathbf{f_0} = \lambda().1;$
	$\texttt{end}\ ();$	$\texttt{exec}\ ();$
$\texttt{let}_s\ b = \{\!\{ \texttt{frag}\%a + 1 \}\!\};$	$\texttt{let}\ b = \texttt{fragment}\ \mathbf{f_1}\ (\texttt{frag}^s\ a);$	$\texttt{bind}\ \mathbf{f_1} = \lambda v.((\texttt{frag}^c\ v) + 1);$
	$\texttt{end}\ ();$	$\texttt{exec}\ ();$
$\texttt{let}_c\ \texttt{return} = \texttt{frag}\%b + 2;$	$\texttt{injection}\ \mathbf{v}\ (\texttt{frag}^s\ b);$	$\texttt{let return} = (\texttt{frag}^c\ \mathbf{v}) + 2;$

Example 5.4: Compilation of expressions

Sections and modules We now present an example with client and server modules in Example 5.5. 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 $\mathrm{ML_S}$ and $\mathrm{ML_C}$ programs is fairly straightforward: Code on the appropriate side has the same shape as in the original $\textsc{Eliom}_\varepsilon$ program and code on the other side only contains calls to the appropriate primitives.

$\textsc{Eliom}_\varepsilon$	$\mathrm{ML_S}$	$\mathrm{ML_C}$
$\texttt{module}_s\ X = \texttt{struct}$	$\texttt{module}\ X = \texttt{struct}$	
$\quad \texttt{let}_s\ a = \{\{\ 2\ \}\}$	$\quad \texttt{let}\ a = \texttt{fragment}\ \mathbf{f_0}\ ()$	$\texttt{bind}\ \mathbf{f_0} = \lambda().2;$
$\quad \texttt{let}_s\ b = 4$	$\quad \texttt{let}\ b = 4$	
$\texttt{end};$	$\texttt{end};$	
	$\texttt{end}\ ();$	$\texttt{exec}\ ();$
$\texttt{module}_c\ F(Y : \mathcal{M}) = \texttt{struct}$		$\texttt{module}\ F(Y : \mathcal{M}) = \texttt{struct}$
$\quad \texttt{let}_c\ a = Y.b + \texttt{int}\%X.b$	$\texttt{injection}\ \mathbf{v_0}\ (\texttt{int}^s\ X.b);$	$\quad \texttt{let}\ a = Y.b + (\texttt{int}^c\ \mathbf{v_0})$
$\texttt{end};$		$\texttt{end};$
$\texttt{module}_c\ Z =$		$\texttt{module}\ Z =$
$\quad F(\texttt{struct let}_c\ b = 2\ \texttt{end});$		$\quad F(\texttt{struct let}\ b = 2\ \texttt{end});$
$\texttt{let}_c\ \texttt{return} =$		$\texttt{let return} =$
$\quad \texttt{frag}\%X.a + Z.a;$	$\texttt{injection}\ \mathbf{v_1}\ (\texttt{frag}^s\ X.a);$	$\quad (\texttt{frag}^c\ \mathbf{v_1}) + Z.a;$

Example 5.5: Compilation of client and server modules and functors

Mixed modules Finally, Example 5.6 presents the compilation of mixed modules. In this example, we create a mixed structure X containing a server declaration and a client declaration with an injection. We define a functor F that takes a module containing a client integer and use it both inside a client fragment, and inside a client declaration. We then apply F to X and use an injection to compute the final result of the program.

The 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 $\texttt{bind}_m$ 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 $\texttt{fragment}_m$ primitive. Each call to $\texttt{fragment}_m$ 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 4.3.3. 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).

ELIOM$_\varepsilon$	ML$_\text{S}$	ML$_\text{C}$
$\texttt{module}_m\ X = \texttt{struct}$ $\quad \texttt{let}_s\ a = 2$ $\quad \texttt{let}_c\ b = 4 + \texttt{int\%}a$ $\texttt{end};$	$\texttt{module}\ X = \texttt{struct}$ $\quad \texttt{module Dyn} = \mathbf{X_0};$ $\quad \texttt{let}\ a = 2;\ \texttt{end}\ ();$ $\quad \texttt{injection}\ \mathbf{v_0}\ (\texttt{int}^s\ a);$ $\texttt{end};$	$\texttt{bind}_m\ \mathbf{X_0} = \texttt{struct}$ $\quad \texttt{exec}\ ();$ $\quad \texttt{let}\ b = 4 + (\texttt{int}^c\ \mathbf{v_0})$ $\texttt{end};$ $\texttt{module}\ X = \mathbf{X_0};$
$\texttt{module}_m\ F(Y : \mathcal{M}) = \texttt{struct}$ $\quad \texttt{let}_s\ c = \{\{\ Y.b\ \}\}$ $\quad \texttt{let}_c\ d = 2 * Y.b$ $\texttt{end};$	$\texttt{module}\ F(Y : \mathcal{M}) = \texttt{struct}$ $\quad \texttt{module Dyn} =$ $\quad\quad \texttt{fragment}_m\ \mathbf{F_1}\ (Y.\texttt{Dyn});$ $\quad \texttt{let}\ c = \texttt{fragment Dyn.}\mathbf{f_0}\ ();$ $\quad \texttt{end}\ ();$ $\texttt{end};$	$\texttt{bind}_m\ \mathbf{F_1}(Y : \mathcal{M}) = \texttt{struct}$ $\quad \texttt{bind}_m\ \texttt{Dyn.}\mathbf{f}_0 = \lambda().(Y.b);$ $\quad \texttt{exec}\ ();$ $\quad \texttt{let}\ d = 2 * Y.b$ $\texttt{end};$ $\texttt{module}\ F(Y : \mathcal{M}) = \texttt{struct}$ $\quad \texttt{let}\ d = 2 * Y.b$ $\texttt{end};$
$\texttt{module}_m\ Z = F(X);$	$\texttt{module}\ Z = F(X);$ $\texttt{end}\ ();$	$\texttt{module}\ Z = F(X);$ $\texttt{exec}\ ()$
$\texttt{let}_c\ \texttt{return} =$ $\quad \texttt{frag\%}Z.c + Z.d;$	$\texttt{injection}\ \mathbf{v_1}\ (\texttt{frag}^s\ Z.c);$	$\texttt{let return} =$ $\quad (\texttt{frag}^c\ \mathbf{v_1}) + Z.d;$

Example 5.6: Compilation of a mixed functor

5.2.1 Sliceability

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.

A program is said sliceable if mixed structures are only defined at top level, or directly inside a toplevel mixed functor. We demonstrate this constraint in Example 5.7. 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.

We discuss this restriction and explain how to partially relax this restriction in Section 5.5.

```
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 5.7: The sliceability constraint

5.2.2 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$_\text{S}$ and the result of $\langle M \rangle_c$ is a module in ML$_\text{C}$.

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 list 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 the rest of this section, we assume that all the considered programs are well typed and sliceable.

We now describe how to slice the various constructions of our language. The slicing rules for modules and expressions are defined in Figure 5.6. The slicing rules for structures and declarations are presented in Figure 5.7.

Base structure and signature components are left untouched. Indeed, according to Proposition 4, base elements are valid ML$_\varepsilon$ 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$_\varepsilon$ type constructs that are valid on the client. Server types, on the other hand, can contains pieces of client types inside fragments $\{\tau_\text{c}\}$ and inside converters $\tau_s \rightsquigarrow \tau_c$. Fragments in ML$_\text{S}$ are represented by a primitive type, `fragment`, without parameters. The type of converters is represented by the type of their server part, which is $\tau_\text{S} \to$ `serial`. Module and module type expressions are traversed recursively.

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

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

Declarations and Definitions

$$\langle \texttt{type}_\iota\ t_i \rangle_\iota = \texttt{type}\ t_i \qquad \langle \texttt{type}_\iota\ t_i = \tau \rangle_\iota = \texttt{type}\ t_i = \langle \tau \rangle_\iota$$
$$\langle \texttt{val}_\iota\ x_i : \tau \rangle_\iota = \texttt{val}\ x_i : \langle \tau \rangle_\iota \qquad \langle \texttt{module}_\varsigma\ X_i : \mathcal{M} \rangle_\iota = \texttt{module}\ X_i : \langle \mathcal{M} \rangle_\iota$$
$$\langle \texttt{let}_\iota\ x_i = e \rangle_\iota = \texttt{let}\ x_i = e \qquad \langle \texttt{module}_\iota\ X_i = M \rangle_\iota = \texttt{module}\ X_i = \langle M \rangle_\iota$$

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

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

Figure 5.6: Slicing – $\langle \cdot \rangle_\iota$

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 new fresh reference noted $\mathbf{v}$. 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 $\mathbf{f}$. 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. We also introduce the decoding part of each converter for escaped values. We then call `exec`, which executes all the pending fragments until the

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

$$\langle D_c; S\rangle_s = (\texttt{injection}\ \mathbf{x_i}\ (f_i^s\ x_i);)_i \langle S\rangle_s \qquad \text{Where } \begin{cases} \overline{f_i\%x_i} = \text{INJS}(D_c) \\ \overline{\mathbf{x}_i} \text{ is a list of fresh variables.} \end{cases}$$

$$\langle D_c; S\rangle_c = \begin{array}{l} D_c\Big[\overline{f_i\%x_i} \mapsto \overline{(f_i^c\ \mathbf{x}_i)}\Big]\ ; \\ \langle S\rangle_c \end{array}$$

$$\langle D_s; S\rangle_s = \begin{array}{l} \langle D_s\rangle_s\Big[\{\!\{\ e_i\ \}\!\}_{\mathbf{f_i}} \mapsto \texttt{fragment}\ \mathbf{f_i}\ \overline{(f_{i,j}^s\ a_{i,j})}\Big]_i\ ; \\ \texttt{end}\ (); \\ \langle S\rangle_s \end{array} \qquad \text{Where } \begin{cases} \overline{\{\!\{\ e_i\ \}\!\}_{\mathbf{f_i}}} = \text{FRAGS}(D_s) \\ \forall i, \overline{f_{i,j}\%a_{i,j}} = \text{INJS}(e_i) \\ \forall i, \overline{x_{i,j}} \text{ are fresh variables} \end{cases}$$

$$\langle D_s; S\rangle_c = \begin{array}{l} \big(\texttt{bind Dyn.}\mathbf{f_i} = \lambda\overline{x_{i,j}}.e_i\big[f_{i,j}\%a_{i,j} \mapsto (f_{i,j}^c\ x_{i,j})\big]\ ;\big)_i \\ \texttt{exec}\ (); \\ \langle S\rangle_c \end{array}$$

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

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

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

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

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

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

Figure 5.7: Slicing of declarations – $\langle D\rangle_\iota$

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 4.3.2. 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 $\mathrm{ML_S}$, 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 $\mathrm{ML_C}$, 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

5.3 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 $\textsc{Eliom}_\varepsilon$ program provides two well typed $\mathrm{ML_C}$ and $\mathrm{ML_S}$ 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 $\textsc{Eliom}_\varepsilon$, 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 $\mathrm{INJS}(D_c) = f\%x$. We note $\mathbf{x}$ the fresh reference. By definition of the typing relation on $\textsc{Eliom}_\varepsilon$, there exists Γ' and τ_c, τ_s such that $\Gamma \subset \Gamma'$, $\Gamma' \triangleright_s f : \tau_s \rightsquigarrow \tau_c$ and $\Gamma' \triangleright_s x : \tau_s$. 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.

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

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

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

$$\dfrac{\dfrac{(\texttt{val}\ f^s : \langle\tau_\mathrm{s}\rangle_\mathrm{s} \to \texttt{serial}) \in \langle\Gamma\rangle_s}{\langle\Gamma\rangle_s \triangleright_{\mathrm{ML_s}} f^c : \langle\tau_\mathrm{s}\rangle_\mathrm{s} \rightsquigarrow \texttt{serial}} \quad \dfrac{}{\langle\Gamma\rangle_s \triangleright_{\mathrm{ML_s}} x : \langle\tau_\mathrm{s}\rangle_\mathrm{s}}}{\langle\Gamma\rangle_s \blacktriangleright_{\mathrm{ML_s}} \texttt{injection}\ \mathbf{x}\ (f^s\ x) : \varepsilon}$$

By induction hypothesis on $\Gamma, (\texttt{val}_c\ x_j : \tau_c) \triangleright_m d_c[f\%x \mapsto x_j] : \varepsilon$ where v_j is fresh, we have
$\langle\Gamma\rangle_c, (\texttt{val}\ x_j : \tau_c) \triangleright_{\mathrm{ML_c}} d_c[f\%x \mapsto x_j] : \varepsilon$. 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.

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

□

5.4 Semantics preservation

We now prove 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 it's 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$_\text{S}$ and ML$_\text{C}$ 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$_\text{S}$ and ML$_\text{C}$ 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

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

5.4.1 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 5.8a is decomposed in Example 5.8b 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.

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 $\forall \alpha_c.(\text{unit} \to \{\alpha_c\}) \rightsquigarrow \alpha_c$).

$$\texttt{let } a = 1 + 2 \texttt{ in}$$
$$\{\{\ 3 + \texttt{int}\%a\ \}\}$$

(a) Fragment with injections

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

(b) Fragment with hoisted injections

Example 5.8: Hoisting on fragments

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

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

and that reductions of the ELIOM$_\varepsilon$ rule CLIENTDECL are always of the following shape:

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

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 5.9a, we can lift the injection out of the functor, as show in Example 5.9b. 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 4.16.

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

$$\texttt{let}_s\ x = \ldots$$
$$\texttt{module}_m\ F(X : \mathcal{M}) = \texttt{struct}$$
$$\quad \texttt{let}_c\ y = f\%x$$
$$\texttt{end}$$

(a) Mixed functor with injections

$$\texttt{let}_s\ x = \ldots$$
$$\texttt{let}_c\ y' = f\%x$$
$$\texttt{module}_m\ F(X : \mathcal{M}) = \texttt{struct}$$
$$\quad \texttt{let}_c\ y = y'$$
$$\texttt{end}$$

(b) Mixed functor with hoisted injections

Example 5.9: Hoisting on mixed modules

5.4.2 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 $\mathrm{ML_S}$ 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.

Remarks about global environments

Let us make some preliminary remarks about global environments in the ELIOM$_\varepsilon$ client generated programs and in $\mathrm{ML_C}$.

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

- Closure fragments, noted $\mathbf{f}$, which come from the execution of `bind env`. The associated value is always a environment (*i.e.*, a signature).
- Fragment values, noted $\mathbf{r}$, 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 $\mathbf{r}$ 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 $\mathrm{ML_C}$. There are only three kind of references:

- Closure fragments, noted $\mathbf{f}$, which come from the slicing of syntactic fragments in the source program. The associated value is always a closure.
- Fragment values, noted $\mathbf{r}$ come from the execution of fragments in the fragment queue.
- Injections, noted $\mathbf{x}$. 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.

Client equivalence

Definition 1 (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 2 (Global environment equivalence)**.** We say that an ELIOM$_\varepsilon$ 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:

    $$e[x_i \mapsto v_i]_i \xrightarrow{\rho \,|\, \gamma \to \gamma}_c v, \theta \implies (\lambda \overline{x}_i.\widehat{\rho}.e\ \widehat{v}_0 \dots \widehat{v}_n) \xrightarrow{|\,\widehat{\gamma} \to \widehat{\gamma}}_{\mathrm{ML_c}} \widehat{v}, \widehat{\theta}$$

    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 3 (Fragment closure environment)**.** We consider that $\widehat{\gamma}_f$ is a fragment closure environment for the ELIOM$_\varepsilon$ 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 FRAGS(e_s) we have $e' = e[f_i \% x_i \mapsto x_i]_i$.

Definition 4 (Functor closure environment)**.** We consider that $\widehat{\gamma}_f$ is a functor closure environment for the ELIOM$_\varepsilon$ 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 ρ, $\widehat{\rho}$, $\gamma = \gamma_f \cup \gamma_r$, $\widehat{\gamma} = \widehat{\gamma}_f \cup \widehat{\gamma}_r \cup \zeta$, e and $\widehat{e}$ such that:

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

Then we have:

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

Proof. The only difference between ELIOM$_\varepsilon$ client expressions and ML$_\mathrm{c}$ expressions are the presence of extra references for injections in ML$_\mathrm{c}$. 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. □

Server equivalence

Definition 5 (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

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

We extend this notation to environments and traces.

5.4.3 Server expressions and structures

We first look at server expressions and structures. By definition of the server reduction relation for ELIOM_ε, the emitted program is a series of binds.

Lemma 2 (Server expressions are simulable)**.** We consider an ELIOM_ε server expression e; the ELIOM_ε 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:

$$e \xRightarrow{\rho_\text{s}}_s v, \mu, \theta_s \qquad\qquad \mu \xRightarrow{\rho_\text{c} \,|\, \gamma \to \gamma'}_c \{\}, \theta_c$$

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

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

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_ε 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:

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

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 $\texttt{fragment}\ \mathbf{f}\ x_1 \dots x_n$.

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

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

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 $\overline{e} \xRightarrow{\rho_c \,|\, \gamma \to \gamma}_c v_c, \theta_c$. 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:

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

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:

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

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

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

with the following invariants:

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

5.4.4 Mixed structures

Lemma 3 (Structures are simulable)**.** We consider a slicable structure S; the $\textsc{Eliom}_\varepsilon$ 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:

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

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

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

with the following invariants:

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

Proof. We consider a slicable structure S; the $\textsc{Eliom}_\varepsilon$ 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:

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

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 $D_b \xRightarrow{\rho_c \,|\, \gamma \to \gamma}_c V_b, \theta_c$ 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 $\langle D_b \rangle_c, \xi \xRightarrow{\rho_c \,|\, \widehat{\gamma} \to \widehat{\gamma}}_{\mathrm{ML_c}} \widehat{V}_c, \xi, \widehat{\theta}_c$ 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:

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

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 $\langle S' \rangle_c, \xi_\bullet +\!\!+ \xi \xRightarrow{\widehat{\rho}_c + \widehat{V}_c \,|\, \widehat{\gamma} \cup \zeta_\bullet \to \widehat{\gamma}'}_{\mathrm{ML_c}} \widehat{V}'_c, \xi, \widehat{\theta}'_c$ for any ξ, with

$$\begin{array}{ccc} \widehat{\gamma}' \simeq^c \gamma' & \widehat{V}'_s \simeq^s V'_s & \widehat{\theta}_s \simeq^s \theta_s \\ FCE(\widehat{\gamma}'_f, V'_s) & \widehat{V}'_c \simeq^c_{\widehat{\gamma}'} V'_c & \widehat{\theta}'_c \simeq^c_{\widehat{\gamma}'} \theta'_c \end{array}$$

We can then build the following derivations:

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

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

and the following invariants are easily verified:

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

• *Case $S = D_s; S'$ – Server declaration.*

We assume that the following executions hold:

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

$$\frac{(\texttt{bind env f}_\mathbf{i})_i \xRightarrow{\rho_c \,|\, \gamma \to \gamma'}_c \{\}, \langle\rangle \qquad \mu \xRightarrow{\rho_c \,|\, \gamma' \to \gamma''}_c V_c, \theta_c \qquad \mu' \xRightarrow{\rho_c \,|\, \gamma'' \to \gamma'''}_c V'_c, \theta'_c}{(\texttt{bind env f}_\mathbf{i})_i; \mu; \mu' \xRightarrow{\rho_c \,|\, \gamma \to \gamma''}_c V_c + V'_c, \theta_c \mathbin{@} \theta'_c}$$

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:

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

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 $\widehat{D}_s = \langle D_s \rangle_s [\{\{\ e_i\ \}\} \mapsto \texttt{fragment}\ \mathbf{f_i}\ \overline{x_{i,k}}]_i$.

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

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

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

Since $(\texttt{bind env}\ \mathbf{f_i})_i \xrightarrow{\rho_c \,|\, \gamma \to \gamma'}_c \{\}, \langle\rangle$, 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:

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

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 $\langle S' \rangle_s \xRightarrow{\widehat{\rho}_s + \widehat{V}_s}_{\mathrm{ML_s}} \widehat{V}'_s, \xi'_\bullet, \zeta'_\bullet, \widehat{\theta}'_c$ and $\langle S' \rangle_c, \xi'_\bullet +\!\!+ \xi' \xRightarrow{\widehat{\rho}_c \,|\, \widehat{\gamma}\text{"} \cup \zeta\text{'}_\bullet \to \widehat{\gamma}\text{"'}}_{\mathrm{ML_c}} \widehat{V}'_c, \xi', \widehat{\theta}'_c$ where

$$\begin{array}{ccc} \widehat{\gamma}''' \simeq^c \gamma''' & \widehat{V}'_s \simeq^s V'_s & \widehat{\theta}'_s \simeq^s \theta'_s \\ FCE(\widehat{\gamma}'''_f, V'_s) & \widehat{V}'_c \simeq^c_{\widehat{\gamma}'} V'_c & \widehat{\theta}'_c \simeq^c_{\widehat{\gamma}'} \theta'_c \end{array}$$

Finally, we can construct the following executions:

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

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

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:

$$\begin{array}{ccc} \widehat{\gamma}''' \simeq^c \gamma''' & \widehat{V}_s + \widehat{V}'_s \simeq^s V_s + V'_s & \widehat{\theta}_s + \widehat{\theta}'_s \simeq^s \theta_s + \theta'_s \\ FCE(\widehat{\gamma}'''_f, V_s + V'_s) & \widehat{V}'_c \simeq^c_{\widehat{\gamma}'''} V'_c & \widehat{\theta}_c + \widehat{\theta}'_c \simeq^c_{\widehat{\gamma}'''} \theta_c + \theta'_c \end{array}$$ ■

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

We assume that the following executions hold:

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

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:

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

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:

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

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 $\overline{D_c} \xrightarrow{\rho_c \,|\, \gamma \to \gamma}_c V_c, \theta_c$. 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 $\downarrow v_i \xrightarrow{|\,\gamma\to\gamma}_c \downarrow v_i, \langle\rangle$. 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 sub-reduction by one of the form $\mathbf{x_i}, \xi \xrightarrow{|\,\widehat{\gamma}'\to\widehat{\gamma}'}_{\mathrm{ML_c}} \downarrow\widehat{v}_c, \xi, \langle\rangle$. for any queue ξ. By Lemma 1, we can build the following reduction:

$$\overline{\widehat{D}_c, \xi \xrightarrow{\widehat{\rho}_c \,|\, \widehat{\gamma}'\to\widehat{\gamma}'}_{\mathrm{ML_c}} \widehat{V}_c, \xi, \widehat{\theta}_c}$$

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 $\langle S'\rangle_c \xrightarrow{\widehat{\rho}_c + \widehat{\mathrm{V}}_c}_{\mathrm{ML_s}} \widehat{V}'_s, \xi'_\bullet, \zeta'_\bullet, \widehat{\theta}'_c$ and $\langle S'\rangle_s, \xi'_\bullet +\!\!+ \xi' \xrightarrow{\widehat{\rho}_s \,|\, \zeta'_\bullet \cup \widehat{\gamma}' \to \widehat{\gamma}''}_{\mathrm{ML_c}} \widehat{V}'_c, \xi', \widehat{\theta}'_c$ where

$$\begin{array}{ccc} \widehat{\gamma}'' \simeq^c \gamma' & \widehat{V}'_s \simeq^s_{\zeta''} V'_s & \widehat{\theta}'_s \simeq^s_{\zeta''} \theta'_s \\ FCE(\widehat{\gamma}''_f, V'_s) & \widehat{V}'_c \simeq^c_{\zeta''} V'_c & \widehat{\theta}'_c \simeq^c_{\zeta''} \theta'_c \end{array}$$

Finally, we can build the following derivations:

$$\frac{\forall i,\ \overline{\texttt{injection}\ \mathbf{x_i}\ x_i \xrightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \varepsilon, [\,], \zeta_\bullet, \langle\rangle} \qquad \overline{\langle S'\rangle_s \xrightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \widehat{V}'_s, \xi'_\bullet, \zeta'_\bullet, \widehat{\theta}'_s}}{(\texttt{injection}\ \mathbf{x_i}\ (f_i^s\ x_i);)_i\ \texttt{end}\ (); \langle S'\rangle_s \xrightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \widehat{V}'_s, \texttt{end} +\!\!+ \xi'_\bullet, \zeta_\bullet \cup \zeta'_\bullet, \widehat{\theta}'_s}$$

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

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:

$$\frac{\dfrac{M \xRightarrow{\rho_s}_m V_s, M^c, \mu, \theta_s}{\texttt{module}_m\ X = M \xRightarrow{\rho_s}_m \{X \mapsto V_s\}, \texttt{module}\ X = M^c; \mu, \theta_s} \quad S' \xRightarrow{\rho_s + \{X \mapsto V_s\}}_m V'_s, \mu', \theta'_s}{\texttt{module}_m\ X = M; S' \xRightarrow{\rho_s}_m \{X \mapsto V_s\} + V'_s, (\mu; \texttt{module}\ X = M^c; \mu'), \theta_s \mathbin{@} \theta'_s}$$

$$\frac{\mu \xRightarrow{\rho_c \mid \gamma \to \gamma'}_c \{\}, \theta_c \quad \dfrac{M^c \xRightarrow{\rho_c \mid \gamma' \to \gamma''}_c V_c, \theta'_c}{\texttt{module}\ X = M^c \xRightarrow{\rho_c \mid \gamma' \to \gamma''}_c \{X \mapsto V_c\}, \theta'_c} \quad \mu' \xRightarrow{\rho_c + \{X \mapsto V_c\} \mid \gamma'' \to \gamma'''}_c V'_c, \theta''_c}{\mu; \texttt{module}\ X = M^c; \mu' \xRightarrow{\rho_c \mid \gamma \to \gamma'''}_c \{X \mapsto V_c\} + V'_c, \theta_c \mathbin{@} \theta'_c \mathbin{@} \theta''_c}$$

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:

$$\begin{array}{lll} \widehat{\gamma}' \simeq^c \gamma'' & \widehat{V}_s \simeq^s V_s & \widehat{\theta}_s \simeq^s \theta_s \\ FCE(\widehat{\gamma}'_f, V_s) & \widehat{V}_c \simeq^c_{\widehat{\gamma}''} V_c & \widehat{\theta}_c \simeq^c_{\widehat{\gamma}'} \theta_c + \theta'_c \end{array}$$

By induction on the execution of S' and μ', we can build the following reduction: $\langle S' \rangle_s \xRightarrow{\widehat{\rho}_s + \left\{X \mapsto \widehat{V}_s\right\}}_{\mathrm{ML_s}} V'_s, \xi'_\bullet, \zeta', \theta'_s$ and $\langle S' \rangle_c, \xi'_\bullet +\!\!+ \xi \xRightarrow{\widehat{\rho}_c + \left\{X \mapsto \widehat{V}_c\right\} \mid \zeta' \cup \widehat{\gamma}' \to \widehat{\gamma}''}_{\mathrm{ML_c}} V'_c, \xi, \theta'_c$, 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:

$$\frac{S \xRightarrow{\rho_s}_m V_s, \mu, \theta_s}{\texttt{struct}\ S\ \texttt{end} \xRightarrow{\rho_s}_m V_s + \{\texttt{Dyn} \mapsto \mathbf{X}\}, \mathbf{X}, \mu, \theta_s}$$

$$\frac{\dfrac{}{\mu_0 \xRightarrow{\rho_c \mid \gamma \to \gamma'}_c V_c, \theta_c} \quad \dfrac{\gamma''(\mathbf{X}) = V_c}{\mathbf{X} \xRightarrow{\rho_c \mid \gamma'' \to \gamma''}_c V_c, \langle\rangle}}{(\texttt{bind}\ \mathbf{X} = \texttt{struct}\ \mu_0\ \texttt{end};\ \texttt{module}\ X = \mathbf{X}) \xRightarrow{\rho_c \mid \gamma \to \gamma''}_c, \theta_c}$$

where $\gamma'' = \gamma' \cup \{\mathbf{X} \mapsto V_c\}$.

We have the following compilations:

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

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

By induction on the execution of S and μ, we have $\langle S \rangle_s \xRightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \widehat{V}_s, \xi_\bullet, \zeta_\bullet, \widehat{\theta}_s$ and $\langle S \rangle_c, \xi_\bullet +\!\!+ \xi \xRightarrow{\widehat{\rho}_c \,|\, \widehat{\gamma} \cup \zeta_\bullet \to \widehat{\gamma}'}_{\mathrm{ML_c}} \widehat{V}_c, \xi, \widehat{\theta}_c$ with the following invariants:

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

We can then build the following executions:

$$\frac{\langle S \rangle_s \xRightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \widehat{V}_s, \xi_\bullet, \zeta_\bullet, \widehat{\theta}_s}{\langle \texttt{module}_m\ X = \texttt{struct}\ S\ \texttt{end}_{\mathbf{X}} \rangle_s \xRightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \left\{ X \mapsto \{\texttt{Dyn} \mapsto \mathbf{X}\} + \widehat{V}_s \right\}, \xi_\bullet, \zeta_\bullet, \widehat{\theta}_s}$$

$$\frac{\dfrac{\langle S \rangle_c, \xi_\bullet +\!\!+ \xi \xRightarrow{\widehat{\rho}_c \,|\, \widehat{\gamma} \cup \zeta_\bullet \to \widehat{\gamma}'}_{\mathrm{ML_c}} \widehat{V}_c, \xi, \widehat{\theta}_c}{\texttt{module}\ X = \texttt{struct}\ \langle S \rangle_c\ \texttt{end}, \xi_\bullet +\!\!+ \xi \xRightarrow{\widehat{\rho}_c \,|\, \widehat{\gamma} \cup \zeta_\bullet \to \widehat{\gamma}'}_{\mathrm{ML_c}} \left\{ X \mapsto \widehat{V}_c \right\}, \xi, \widehat{\theta}_c}}{\langle \texttt{module}_m\ X = \texttt{struct}\ S\ \texttt{end} \rangle_c, \xi_\bullet +\!\!+ \xi \xRightarrow{\widehat{\rho}_c \,|\, \widehat{\gamma} \cup \zeta_\bullet \to \widehat{\gamma}''}_{\mathrm{ML_c}} \left\{ X \mapsto \widehat{V}_c \right\}, \xi, \widehat{\theta}_c}$$

Where $\widehat{\gamma}'' = \widehat{\gamma}' \cup \left\{ \mathbf{X} \mapsto \widehat{V}_c \right\}$. We verify the following invariants:

$$\begin{array}{ccc} \widehat{\gamma}'' \simeq^c \gamma' & \widehat{V}_s + \{\texttt{Dyn} \mapsto \mathbf{X}\} \simeq^s V_s + \{\texttt{Dyn} \mapsto \mathbf{X}\} & \widehat{\theta}_s \simeq^s \theta_s \\ FCE(\widehat{\gamma}''_f, V_s) & \widehat{V}_c \simeq^c_{\widehat{\gamma}'} V_c & \widehat{\theta}_c \simeq^c_{\widehat{\gamma}'} \theta_c \end{array}$$

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 $\mu = \texttt{bind env}\ \mathbf{F}$ and the module expression

$M^c = \texttt{functor}(X_i : \mathcal{M}_i)_i \texttt{struct}\ S|_c\ \texttt{end}$. The following reductions hold:

$$\frac{}{\left(\begin{array}{l}\texttt{module}_m\ F(X_i : \mathcal{M}_i)_i = \texttt{struct}\\ \quad S\\ \texttt{end}_{\mathbf{F}}\end{array}\right) \xRightarrow{\rho_s}_m \{F \mapsto V_s\}, \left(\begin{array}{l}\texttt{bind env } \mathbf{F}\\ \texttt{module}_m\ F(X_i : \mathcal{M}_i)_i = \texttt{struct}\\ \quad S|_c\\ \texttt{end}\end{array}\right), \langle\rangle}$$

$$\frac{}{\left(\begin{array}{l}\texttt{bind env } \mathbf{F}\\ \texttt{module}_m\ F(X_i : \mathcal{M}_i)_i = \texttt{struct}\\ \quad S|_c\\ \texttt{end}\end{array}\right) \xRightarrow{\rho_c \,|\, \gamma \to \gamma'}_c \{F \mapsto V_c\}, \langle\rangle}$$

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:

$$\frac{}{\left\langle \begin{array}{l}\texttt{module}_m\ F(X_i : \mathcal{M}_i)_i = \texttt{struct}\\ \quad S\\ \texttt{end}_{\mathbf{F}}\end{array} \right\rangle_s \xRightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \left\{F \mapsto \widehat{V}_s\right\}, \xi_\bullet, \{\}, \langle\rangle}$$

$$\frac{}{\left\langle \begin{array}{l}\texttt{module}_m\ F(X_i : \mathcal{M}_i)_i = \texttt{struct}\\ \quad S\\ \texttt{end}_{\mathbf{F}}\end{array} \right\rangle_c, \xi \xRightarrow{\widehat{\rho}_c \,|\, \widehat{\gamma} \to \widehat{\gamma}'}_{\mathrm{ML_c}} \left\{F \mapsto \widehat{V}_c\right\}, \xi, \langle\rangle}$$

Where $\gamma' = \gamma \cup \left\{\mathbf{F} \mapsto \widehat{V}_{\mathbf{F}}\right\}$ with the following values:

$$\begin{aligned}
\widehat{V}_s &= \texttt{functor}(\widehat{\rho}_s)(X_i : \mathcal{M}_i)_i \texttt{struct } \langle S \rangle_s \texttt{ end} \\
\widehat{V}_c &= \texttt{functor}(\widehat{\rho}_c)(X_i : \mathcal{M}_i)_i \texttt{struct } S|_c \texttt{ end} \\
\widehat{V}_{\mathbf{F}} &= \texttt{functor}(\widehat{\rho}_c)(X_i : \mathcal{M}_i)_i \texttt{struct } \langle S \rangle_c \texttt{ end}
\end{aligned}$$

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:

$$\frac{\begin{array}{l}
F \xRightarrow{\rho_s}_m \texttt{functor}_m(\rho'_s)(Y_i : \mathcal{M}_i)_i \texttt{struct } S \texttt{ end}_{\mathbf{F}}, \varepsilon, \langle\rangle \qquad X_i \xRightarrow{\rho_s}_m V_i^s, \varepsilon, \langle\rangle \\
V_i^s(\texttt{Dyn}) = \mathbf{R_i} \qquad \mathbf{R} \text{ fresh} \qquad S[\mathbf{f_i} \mapsto \mathbf{R}.\mathbf{f_i}]_i \xRightarrow{\rho'_s + \left\{Y_i \mapsto V_i^s\right\}_i}_m V, \mu_0, \theta
\end{array}}{F(X_1)\dots(X_n) \xRightarrow{\rho_s}_m V_s + \{\texttt{Dyn} \mapsto \mathbf{R}\}, \mu = \begin{pmatrix} \texttt{bind } \mathbf{R} = \texttt{struct} \\ \quad (\texttt{module } Y_i = \mathbf{R_i};)_i \\ \quad \mu_0 \\ \texttt{end with } \mathbf{F} \end{pmatrix}, \theta}$$

$$\overline{\texttt{module}_m\ X = F(X_1)\dots(X_n) \xRightarrow{\rho}_m V_s, (\mu; \texttt{module } X = F(X_1)\dots(X_n)), \theta_s}$$

$$\frac{\gamma(\mathbf{F}) = \rho_{\mathbf{F}} \qquad \gamma(\mathbf{R_i}) = V_i^c \qquad \mu_0 \xrightarrow{\rho_{\mathbf{F}} + \left\{ \mathrm{Y_i} \mapsto \mathrm{V_i^c} \right\}_{\mathrm{i}} \mid \gamma \to \gamma'}_c V_c, \theta_c}{\begin{pmatrix} \texttt{bind } \mathbf{R} = \texttt{struct} \\ \quad (\texttt{module } Y_i = \mathbf{R_i};)_i \\ \quad \mu_0 \\ \texttt{end with } \mathbf{F} \end{pmatrix} \xrightarrow{\rho_{\mathrm{c}} \mid \gamma \to \gamma' \cup \{\mathbf{R} \mapsto \mathrm{V_c}\}}_c \varepsilon, \theta_c}$$

$$\frac{\begin{array}{l} F \xrightarrow{\rho_{\mathrm{s}} \mid \cdots \to \cdots}_c \texttt{functor}(\rho_c')(Y_i : \mathcal{M}_i)_i \texttt{struct } S_c \texttt{ end}, \langle\rangle \\ X_i \xrightarrow{\rho_{\mathrm{c}} \mid \cdots \to \cdots}_c V_i, \langle\rangle \qquad S_c \xrightarrow{\rho'_{\mathrm{c}} + \{\mathrm{Y_i} \mapsto \mathrm{V_i}\}_{\mathrm{i}} \mid \gamma' \cup \{\mathbf{R} \mapsto \mathrm{V_c}\} \to \gamma''}_c V_c', \theta_c' \end{array}}{\texttt{module } X = F(X_1) \ldots (X_n) \xrightarrow{\rho_{\mathrm{c}} \mid \gamma' \cup \{\mathbf{R} \mapsto \mathrm{V_c}\} \to \gamma''}_c \left\{ X \mapsto V_c' \right\}, \theta_c'}$$

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:

$$\langle \texttt{module}_m \ X = F(X_1) \ldots (X_n) \rangle_s = \begin{array}{l} \texttt{module } X = F(X_1) \ldots (X_n); \\ \texttt{end } (); \end{array}$$

$$\langle \texttt{module}_m \ X = F(X_1) \ldots (X_n) \rangle_c = \begin{array}{l} \texttt{module } X = F(X_1) \ldots (X_n); \\ \texttt{exec } (); \end{array}$$

Let us now look at the execution of the server application $F(X_1) \ldots (X_n)$. 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 $\widehat{V}_i^s = \widehat{\rho}_s(X_i)$. By equivalence, we have $\widehat{V}_i^s \simeq^s V_i^s$.

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 $V_i^c = \widehat{\gamma}(\mathbf{R_i})$. By equivalence, we have $\widehat{V}_i^c \simeq^s V_i^c$.

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 $\widehat{\rho}_s \cup \{\texttt{Dyn} \mapsto \mathbf{R}\} \cup \left\{ Y_i \mapsto \widehat{V}_i^s \right\}_i$. We obtain the following reductions:

$\langle S \rangle_s \xrightarrow{\widehat{\rho}'_{\mathrm{s}} \cup \left\{ \mathrm{Y_i} \mapsto \widehat{\mathrm{V}}_{\mathrm{i}}^{\mathrm{s}} \right\}_{\mathrm{i}}}_{\mathrm{ML_s}} \xi_\bullet, \zeta_\bullet, \widehat{\theta}_s$, and $\langle S \rangle_c, \xi_\bullet +\!\!+ \xi \xrightarrow{\widehat{\rho}_{\mathrm{c}} \cup \{\mathrm{Y_i} \mapsto \mathbf{R_i}\}_{\mathrm{i}} \mid \zeta_\bullet \cup \widehat{\gamma} \to \widehat{\gamma}'}_{\mathrm{ML_c}} \widehat{V}_c, \xi, \widehat{\theta}_c$

with the usual invariants. We can now build the following executions:

$$\frac{\texttt{fragment}_m\ \mathbf{F}\ (Y_i.\texttt{Dyn})_i \xrightarrow{\widehat{\rho}'_s \cup \left\{Y_i \mapsto \widehat{V}_i^s\right\}_i}_{\mathrm{ML_s}} \mathbf{R}, \xi_{\mathbf{R}}, \varepsilon, \langle\rangle \qquad \widehat{\rho}_s(F) = \widehat{V}_f \qquad \widehat{\rho}_s(X_i) = \widehat{V}_i^s \qquad \langle S \rangle_s \xrightarrow{\widehat{\rho}'_s \cup \{\texttt{Dyn} \mapsto \mathbf{R}\} \cup \left\{Y_i \mapsto \widehat{V}_i^s\right\}_i}_{\mathrm{ML_s}} \xi_\bullet, \zeta_\bullet, \widehat{\theta}_s,}{\texttt{module}\ X = F(X_1) \ldots (X_n); \texttt{end}\ () \xrightarrow{\widehat{\rho}_s}_{\mathrm{ML_s}} \left\{X \mapsto \widehat{V}_s\right\}, \xi_{\mathbf{R}} +\!\!+ \xi_\bullet +\!\!+ \texttt{end}, \zeta_\bullet, \widehat{\theta}_s}$$

$$\frac{\widehat{\gamma}(\mathbf{F}) = \widehat{V}_{\mathbf{F}} \qquad \langle S \rangle_c, \xi_\bullet +\!\!+ \texttt{end} +\!\!+ \xi \xrightarrow{\widehat{\rho}_{\mathbf{F}} \cup \{Y_i \mapsto \mathbf{R_i}\}_i \,|\, \zeta_\bullet \cup \widehat{\gamma} \to \widehat{\gamma}'}_{\mathrm{ML_c}} \widehat{V}_c, \texttt{end} +\!\!+ \xi, \widehat{\theta}_c}{\texttt{exec}\ (), \xi_{\mathbf{R}} +\!\!+ \xi_\bullet +\!\!+ \texttt{end} +\!\!+ \xi \xrightarrow{\widehat{\rho}_c \,|\, \zeta_\bullet \cup \widehat{\gamma} \to \widehat{\gamma}' \cup \left\{\mathbf{R} \mapsto \widehat{V}_c\right\}}_{\mathrm{ML_c}} \varepsilon, \xi, \widehat{\theta}_c}$$

where $\xi_{\mathbf{R}} = \{\mathbf{R} \mapsto \mathbf{F}(\mathbf{R_1}) \ldots (\mathbf{R_n})\}$. We respect the following invariants:

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

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:

$$\frac{\widehat{\rho}_c(F) = \texttt{functor}(\widehat{\rho}'_c)(Y_i : \mathcal{M}_i)_i \texttt{struct}\ \widehat{S}_c\ \texttt{end} \qquad \widehat{\rho}_c(X_i) = \widehat{V}_i^c \qquad \widehat{S}_c \xrightarrow{\widehat{\rho}'_c + \left\{Y_i \mapsto \widehat{V}_i^c\right\}_i \,|\, \widehat{\gamma}' \cup \left\{\mathbf{R} \mapsto \widehat{V}_c\right\} \to \widehat{\gamma}''}_{\mathrm{ML_c}} \widehat{V}'_c, \widehat{\theta}'_c}{\texttt{module}\ X = F(X_1) \ldots (X_n), \xi \xrightarrow{\widehat{\rho}_c \,|\, \widehat{\gamma}' \cup \left\{\mathbf{R} \mapsto \widehat{V}_c\right\} \to \widehat{\gamma}''}_{\mathrm{ML_c}} \left\{X \mapsto \widehat{V}'_c\right\}, \xi, \widehat{\theta}'_c}$$

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:

$$\frac{\begin{array}{l}\texttt{exec}\ (), \{\mathbf{R} \mapsto \mathbf{F}(\mathbf{R_1}) \ldots (\mathbf{R_n})\} +\!\!+ \xi_\bullet +\!\!+ \texttt{end} +\!\!+ \xi \xrightarrow{\widehat{\rho}_c \,|\, \zeta_\bullet \cup \widehat{\gamma} \to \widehat{\gamma}' \cup \left\{\mathbf{R} \mapsto \widehat{V}_c\right\}}_{\mathrm{ML_c}} \varepsilon, \xi, \widehat{\theta}_c \\ \texttt{module}\ X = F(X_1) \ldots (X_n), \xi \xrightarrow{\widehat{\rho}_c \,|\, \widehat{\gamma}' \cup \left\{\mathbf{R} \mapsto \widehat{V}_c\right\} \to \widehat{\gamma}''}_{\mathrm{ML_c}} \left\{X \mapsto \widehat{V}'_c\right\}, \xi, \widehat{\theta}'_c\end{array}}{\langle \texttt{module}_m\ X = F(X_1) \ldots (X_n) \rangle_c, \xi_{\mathbf{R}} +\!\!+ \xi_\bullet +\!\!+ \texttt{end} +\!\!+ \xi \xrightarrow{\widehat{\rho}_c \,|\, \zeta_\bullet \cup \widehat{\gamma} \to \widehat{\gamma}''}_{\mathrm{ML_c}} \left\{X \mapsto \widehat{V}'_c\right\}, \xi, \widehat{\theta}_c \mathbin{@} \widehat{\theta}'_c}$$

where the invariants still hold. This concludes. ■

□

5.4.5 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:

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

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. □

5.5 Discussion around mixed functors

In this thesis, we presented the notion of "mixed functors", which are functors that take as argument and return mixed module composed of both client and server code. As we saw, those functors, while quite expressive, have several limitations. In this section, we will try to explore a little bit the design space around mixed functors and which limitations we think could be lifted.

First, let us recall the design constraints of Eliom: typing and slicing are done statically and separate compilation is supported. This prevents us from "erasing" functors by inlining them and also prevents us from dynamically generating the code contained in functors. Furthermore, we want to support languages extensions such as OCaml, where functor application can depend on the control flow (notably, first class modules).

One alternative solution would be *fully separable* functors: mixed functors such that client and server execution are completely independent. While this would be easy to implement, it would also mean preventing any meaningful usage of fragments inside functors. Our version of mixed functors is slightly more expressive: the client part of the functor is indeed independent from the server part, but the server part is not. The cost is that we must do some extra book-keeping to ensure that for each server-side application of a mixed functors, all the client side effects are performed. We believe this expressive power is sufficient for most use cases. There are however several limitations to our approach, which we shall discuss point-by-point now.

Mixed functors arguments An important restriction of mixed functors is that their arguments can only be mixed structures. The reason for this restriction is that the `Dyn` field is used in order to keep track of client applications from the server. In the semantics presented in this thesis, the `Dyn` field is represented simply as a regular structure field, which means only structures can be passed as argument. A first step would be to allow a similar field to be added to functors. While it is a bit delicate to formalize, it should be possible to implement it in OCaml by simply adding an additional field to the closure. A second step would be to allow base, client and server modules as arguments. As with usual mixed functors, care must be taken during typechecking to not specialize eagerly. Given that this constraint is respected, one possibility would be to introduce a new module-level operation that can take a base, server or client structure and turn it into a mixed structure. Given a client module `A`, this could be done by inserting a structure of the form `(struct include%client A end)`. This could even be done transparently, since the location of modules is always known.

Injections Injections inside mixed functors can only refer to identifiers outside of the functor. This restriction seems particularly difficult to lift, as demonstrated by the example in Section 4.2.2. One would need to evaluate the usefulness of mixed functors in first-class contexts. It might be possible to rewrite usages of injections to escaped values in fragments. In all cases, these changes would be invasive and of limited usefulness.

Sliceability constraints The sliceability requirement presented in Section 5.2.1 is quite restrictive. Its goal is to ensure that $\texttt{bind}_m$ expressions are not nested and that module references are unique. This requirement can be relaxed in different manners. For mixed structures without functors, this requirements can be completely removed trivially, since internal modules are known.

For functors, there are two possibilities. A first idea would be to apply lambda-lifting to functors. By simply lifting mixed functors to the outer scope, we ensure that `bind` operations are noted nested. Another possibility would be to use a similar technique to the one used for fragments inside functors: by prefixing each statically generated reference with the locally available `Dyn` field, we ensure uniqueness while allowing arbitrary nesting.

Double application of the client side of mixed functors As mentioned in Section 4.3.3, the client-side part of a mixed functor, when applied in a mixed context, might be called twice. While this is not so problematic in the context of applicative and pure functors, it might prove more troublesome with generative or impure functors. This problem is difficult to solve in general. In particular, it is in fairly direct conflict with the design decision, justified by Section 4.2.2, to make the client and server side of a mixed functor application independent. One potential solution would be to provide a special interpretation of mixed application in mixed contexts that would ensure that the result of the client-side functor application is properly reused. Notably, this might be doable by generating at compile time an identifier for each known returned client module.

6 Implementation

> *Its black gates are guarded by more than just orcs. There is evil there that does not sleep. The great eye is ever watchful. It is a barren wasteland, riddled with fire, ash, and dust. The very air you breathe is a poisonous fume.*
>
> J. R. R. Tolkien, *The Fellowship of the Ring*

The main goal of the formalization of ELIOM$_\varepsilon$ was to inform the design and the implementation of ELIOM as a real extension of OCAML usable to develop real Web applications. As a consequence, everything presented in Chapter 4 except for mixed functors is also implemented in a fork of the OCAML compiler. This fork supports the complete OCAML language. OCAML is quite a large language and a lot of aspects of a real programming language are not covered by the formal specification.

This chapter presents how ELIOM is implemented, the various design decisions that were made related to compilation, interaction with OCAML and tooling, along with the few compromises that were occasionally made. Some sections are quite technical and assume knowledge of the inner workings of the OCAML compiler.

Everything developed in relation to this thesis is published as free software[1] under the GPL and LGPL licence (with the OCAML linking exception):

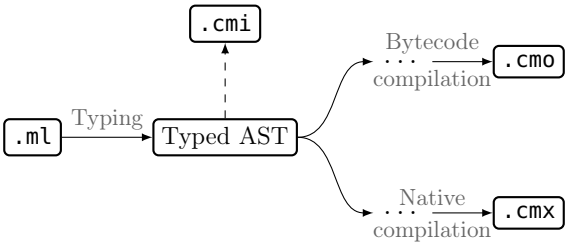
- `https://github.com/ocsigen/ocaml-eliom` is the fork of the compiler;
- `https://github.com/ocsigen/eliomlang` contains the ELIOM runtime and a collection of associated tools.

6.1 The OCaml compiler

Let us start by a quick reminder of the architecture of the OCAML compiler. The compiler pipeline is presented in Figure 6.1a. Figure 6.1b summarizes the numerous files involved in the OCAML compilation. The C files are also presented, for comparison.

Due to separate compilation and typechecking, the OCAML compiler uses many files. `.cmi` files are compiled versions of interfaces, or module types. They are used for the purpose of separate compilation: when subsequent compilations need an already compiled module, the compiler loads the file in the typing environment. Bytecode (resp. Native) compilation of a module produces a `.cmo` (resp. `.cmx`) object file. Several of these files can be grouped together in libraries as `.cma` (resp. `.cmxa`) files. Compiled objects and libraries can be linked to produce a bytecode or native executable. The compiler also

[1] I would certainly not deserve Roberto as my advisor otherwise!



(a) The OCaml pipeline

	Byte	Native	C
Sources	.ml		.c
Interfaces	.mli		.h
Compiled Interfaces	.cmi		
Compiled Modules	.cmo	.cmx	.o
Compiled Libraries	.cma	.cmxa	.a

(b) The OCaml file extensions

Figure 6.1: The OCaml compilation

produces several other files such as `.cmt` and `.cmti` for documentation and `.cmxs` for inlining.

The pipeline is composed of three parts: bytecode compilation, native compilation and a common part, which we simply call "frontend", composed of parsing and typechecking. An interface source file (`.mli`) is the text format for module types. After parsing and verification of well-formedness, it is transformed into a compiled interface and produces a standalone `.cmi` file. A source file (`.ml`) is treated in several steps. Parsing produces an abstract syntax tree (or AST) with no typing information. Typechecking produces a Typed AST and a compiled interface. The typed AST is an AST where all the types have been made completely explicit: each node of the AST is annotated with its type and its local environment. The interface is the inferred module type of the file. If a corresponding module type is present (as a `.mli` file), inclusion of the two interfaces is checked. If no module type is present, a `.cmi` file is produced. After typechecking, the compiler proceeds to bytecode or native code generation. For our purpose, everything after typechecking is a black box that produces the desired compiled files.

6.2 The Eliom compiler

We now look at our modified Eliom compiler. The pipeline for OCaml file is identical to the regular pipeline. The pipeline for Eliom file is presented in Figure 6.2.

Eliom files are either `.eliom` or `.eliomi`, which corresponds to `.ml` and `.mli` respectively. The pipeline is similar to the regular OCaml one. Each Eliom file is first typechecked using a modified typechecker presented in Section 6.4. This creates an Eliom typed AST, which still contains both side of the program. A unique compiled interface (`.cmi`) is created. We then slice the typed AST according to the procedure presented in Section 5.2, with returns two untyped AST, one for the client part and one for the server part. After slicing, we obtain two pure OCaml programs that can be compiled with the regular OCaml compiler which return two compiled objects `.server.cm[ox]` and `.server.cm[ox]`. The ML_S and ML_C primitives introduced in Section 5.1 are implemented in two libraries which are presented in Section 6.5.

One might wonder why the slicing is implemented in such a convoluted way: we first

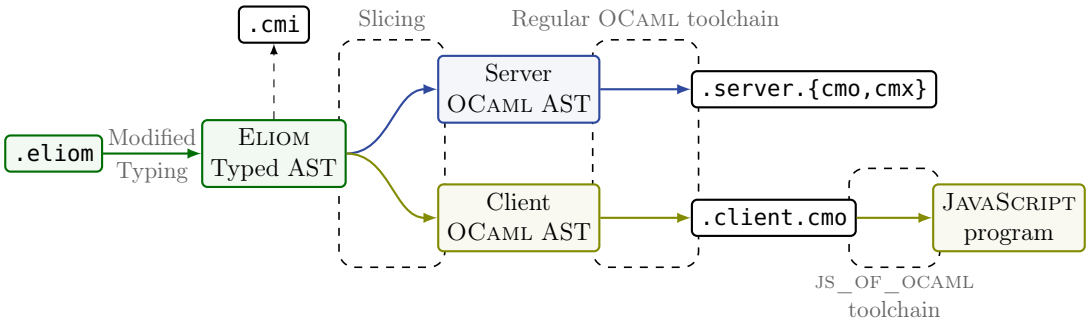


Figure 6.2: The Eliom pipeline

type using a modified typechecker, then slices on the typed program and return two *untyped* programs which are typed again using the regular typechecker. The first reason is safety: By using the regular typechecker, we increase trust in our modified typechecker. The second reason is more pragmatic: The OCaml typed AST is a complex datastructure which is highly optimized for fast type inference and contains non trivial invariants. Doing transformations on such datastructure while preserving its invariants is a difficult task. Emission of untyped AST, on the other hand, is well supported. Thanks to the PPX ecosystem, numerous tools are available and the compiler is robust against ill-formed ASTs which makes developing such transformations easier.

6.2.1 Notes on OCaml compatibility

One of the important results on the Eliom module system given Section 4.4.1 is that any vanilla OCaml module is also an Eliom$_\varepsilon$ module, located either on base, client or server. This is also true for the implementation. We can even go further.

Compiled interfaces created by the regular compiler can be loaded by the Eliom compiler. A compiler flag allows to specify on which location to load the given interfaces. Compiler interfaces created by the Eliom compiler, which contains extra location information, cannot be loaded by the OCaml compiler (as it would not be able to make sense of them). Server and Client object files are compatible with the regular compilers (and can be linked together with object files emitted by the regular compiler). The consequence is that the OCaml and Eliom compilers can live graciously side by side and any OCaml library can be used directly in Eliom.

The various files used by external tools (such as `.cmt` files) are also similarly compatible.

6.3 Converters

In the formalization, injections are modeled as the application of a converter; *i.e.*, a pair of a serialization and a deserialization function; to a value available on the server (See Section 4.2.1). Furthermore, converters are global constants introduced by the typing

environment. This formalization, however, is inconvenient to program with: it would mean the programmer needs to specify which converter to use for each injection, even the most trivial ones. We also need a way to define new converters easily. One promising idea would be to use ad-hoc polymorphism to infer which converter to use based on type information. We explore this lead in Section 6.3.1 and how it interacts with our module system. Unfortunately, ad-hoc polymorphism is not yet available in OCaml. In Section 6.3.2 we present what is currently implemented in Eliom instead.

6.3.1 Modular implicits

Modular implicits [White et al., 2014] are an extension of OCaml that adds ad-hoc polymorphism in a way that is compatible with OCaml's module system. An introduction to modular implicits can be found in White et al. [2014]. The main idea is that we can use modules themselves as implicit arguments. Converters would then be mixed modules satisfying the signature `CONV` presented in Figure 6.3. The `String` module satisfies the `CONV` signature and exports that the client and the server type are equal to the base type `string`. We can also specify converters for parametrized datatypes using functors, as presented by the `List` mixed functor. The injection operator `~%` will then take an implicit module `C` satisfying the `CONV` signature, take an element of type `C.t` *on the server* and returns an element of type `C.t` *on the client*. Finally, we can also easily implement the `frag` converter by using a functor taking a module with just a datatype. Note that we only need a client datatype and no serialization method.

This implementation has several advantages. First, it means that injections are properly parametric: Inference will work on implicit modules in parametric functions and it plays well with polymorphism. Additionally, defining a new converter is very easy: it is simply a module respecting a fairly simple signature. Finally, the projection functions f^s and f^c used in the target languages $\mathrm{ML_S}$ and $\mathrm{ML_C}$ (Section 5.1.1) are very easy to define: given a converter module `M`, they are `M.serialize` and `M.deserialize`. Note that implementing such a signature is not completely trivial. We discuss serialization techniques in Section 6.5.2.

6.3.2 Wrapping

Unfortunately, at the time of writing of this thesis, modular implicits are not yet integrated in OCaml. Eliom uses instead a method that was originally devised by Pierre Chambart and Gregoire Henry and relies on value introspection. The main idea is the following: most values can be serialized simply by using the module `Marshal` from the OCaml standard library [marshal]. As shown on multiple occasions in Chapter 2, it can be useful to transform data before serializing it. In this case, a transformation function is attached to the value. Before serializing, the function is applied to the value and the result is serialized. For customized deserialization, a similar technique is applied using tags and a client-side association table from tag to functions.

While this has the advantage of leveraging `Marshal`, which works for any data structure, is very fast and preserves sharing, it also has significant disadvantages. First, it is

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

implicit%mixed String : CONV
  with type%server t = string
   and type%client t = string

implicit%mixed List {M : CONV} : CONV
  with type%server t = M.t list
   and type%client t = M.t list

implicit%mixed Fragment {M : sig type%client t end} : CONV
  with type%server t = M.t fragment
   and type%client t = M.t

val%client (~%) : {C : CONV} -> C.t(*server*) -> C.t(*client*)
```

Figure 6.3: A signature for converters

not type-directed. Closures for example cannot be serialized, which will not be detected by the type system. Furthermore, defining converters is quite delicate: the programmer must create a function operating on values in a fairly untyped way and attach it to the value being converted. This is an expert task that is very error-prone.

6.4 Typechecking and Slicing

Most of the differences between the Eliom compiler and the OCaml compiler are modifications of the typechecker. A formal account of the difference between the two type systems was given in Section 4.2. We now give a practical account of these differences and explain how the OCaml type checker was modified.

The OCaml language is much larger than our ML language. In particular it contains intrusive features such as first class modules and Generalized Algebraic Datatypes which affect many parts of the language. Our modified typechecker handles most of these features correctly thanks to the fact that most additional typing rules for Eliom are at the boundaries between locations. By being conservative in the type conversions across location boundaries (in particular, by prohibiting existential and universal types), we avoid difficulties related to complex features of the OCaml type system. In other aspects, the implementation follows the formalization closely: there are now distinct namespaces for each location, with visibility rules that correspond to the relations presented in Section 4.1.1. The use of explicit locations allows to provide good error messages for missing identifiers and wrongly located declarations such as client declarations inside server modules.

One notable new feature of the implementation is the inference of some location in-

formation. In particular declarations inside base, client and server modules do not need any annotations. Similarly, most mixed annotations on modules can be elided. These elisions are based on the fact that in those cases, declaration can only have one location, which make it superfluous.

Shared declarations Another significant addition is the **`shared`** annotation. This annotation has no equivalent in the formalization. In particular, it does *not* correspond to mixed modules. Shared sections were presented in Section 2.4.4 and are implemented simply by duplicating the code between client and server, as demonstrated in Example 6.1. Note that this is a purely syntactic transformation done before typechecking. From a typechecking perspective, `y` is not a singular variable to which can be assigned a type. There are two variables, one client and one server, with two different types. This is different from `mixed` annotation where a structure contains both client and server parts. This slightly schizophrenic nature of shared declarations does not lead itself to being internalized in the type system. Nevertheless, it avoids repetitive code patterns where a similar tasks is done both client and server side with a few distinct values, which makes it very useful in practice.

```
1 let%client x = 2
2 let%server x = "foo"
3 let%shared y = x
4
```

(a) Original code

```
1 let%client x = 2
2 let%server x = "foo"
3 let%client y = x
4 let%server y = x
```

(b) Desugared code

Example 6.1: A shared declaration

Slicing Slicing is also identical to the formalization. An example of sliced programs is given in Example 6.2. References are represented by unique generated strings.

A notably difficult point is the generation of fresh identifiers. Indeed, in our slicing formalization, we rely on generating fresh identifiers for each injection and each fragment closure. These identifiers should be globally unique in the complete program. However,

	Server	Client
`let%client c =` `  ~%s + 1`	`push_injection "A.s1" s`	`let c = get_injection "A.s1" + 1`

`let%server y =`
`  [%client 2 + ~%x ]`
	`let y =`
	`  fragment "B1" (x)`
	`push_fragments "B"`
		`register_closure "B1"`
		`  (fun x -> 2 + x)`
		`execute_fragments "B"`

Example 6.2: Client-server code slicing

compilation units are sliced separately. The list of identifiers generated by slicing other modules is not available. To overcome this limitation, we rely on the property that, in a given OCaml program, top level module names are unique. This property is enforced by the OCaml compiler at link time. Consequently, we can generate locally unique identifiers prefixed by the current module path, which produces globally unique name in a way that is compatible with separate compilation.

6.4.1 Technical details

We now present a technical account of the modifications made to the compiler. This section is targeted at ambitious readers who are willing to gaze into the typechecker[2] in order to modify or improve the Eliom language. We assume that the reader is familiar with the implementation of the OCaml typechecker. For a more gentle introduction to the typechecker, please consult the file `typing/HACKING.adoc` in the standard OCaml distribution.

The typechecker is both complex and frequently updated. Consequently, changes to the implementation must be made with the smallest footprint possible, in order to avoid conflicts with future modifications. One of the important early design decision for the Eliom compiler was complete compatibility with standard OCaml tools. OCaml files format such as `.cmi` correspond to serializations of the internal data structure of the compiler (notably, the typedtree). Any changes to the typedtree, type environment or representation of type expressions would lead to changes in the compiler files which would break binary compatibility. This leads to the following programming rule: "Thou shall not change data structures", of which we will now explore the consequences.[3] Some techniques used by this implementation are partially inspired by the MetaOCaml patches[4]. MetaOCaml [Kiselyov, 2014] is an extension of OCaml for meta-programming which employs quotations and slicing constructions that are similar to Eliom's fragments and injections. Notably, MetaOCaml manages to add a quotation mechanism to the OCaml type system in a remarkably lean way (less than 1000 lines of changes on existing files).

Locations, fragments and injections The first task the modified typechecker need to do is to keep track of the local location of the considered code. In order to avoid passing this information around by additional arguments, we simply use a global reference. While this is not particularly elegant, there are various precedents in the OCaml typechecker (levels, notably). One additional difficulty is that the typechecker occasionally uses exception for control flow, which means we need to ensure that our global reference is put back in order. For this purpose, we use the traditional functional pattern of closure-based handler. While this approach sacrifices tail-recursion, it has not proved problematic on concrete programs.

```
1 val in_location : location -> (unit -> 'a) -> 'a
```

[2] No, it does not gaze back. Typecheckers do not gaze. They do however inflict 2d6 sanity damage on failed bootstraps.

[3] Naturally, as demonstrated by the Oulipo, programming constraints only encourage creativity.

[4] http://okmij.org/ftp/ML/MetaOCaml.html

In order to typecheck injections and fragments, we simply leverage `in_location` to typecheck the inner expression and use unification to reflect the inferred type upward. Unfortunately, it is not possible to introduce additional nodes in the typedtree, but it is possible to add extra PPX annotations. Thus we simply return the typedtree of the inner expression with custom annotations (`[@eliom.fragment]` and `[@eliom.injection]`, notably). We then use these annotations to drive the slicing phase.

Note that `in_location` is not only used for typechecking. Indeed, locations must also be tracked when walking type expressions for validation and unification due to the presence of mixed datatypes (Section 4.2.1).

Identifiers and bindings In the formalization, bindings are annotated with location information. This is, unfortunately, difficult to achieve in the current implementation of the typechecker without large changes to the `Env` module. An alternative solution is used: instead of annotating binders, we annotate identifiers. Identifiers are represented by the `Ident.t` datatype, whose definition is given in Figure 6.4. The name field is the name of the identifier given in the source file. The stamp is a unique identifier. The flag field is a bitfield collecting certain flags necessary for the compiler. Only two bits are used in this integer (one to recognize top level module names, one for built in exceptions) which leaves at least 29 free bits. Fortunately, we can use this free space to store locations information! The Eliom compilers then uses two additional bits which we will call "c" and "s" in order to determine the location of a given identifier according to the table presented Figure 6.5. The typechecker maintains a set of association tables of identifiers to values. This is implemented by `Ident.tbl`, which implements lookup by identifiers, stamp but also lookup by name. We re-purpose these lookup functions by making them respect the "can be used" relation (Figure 4.2). This can be achieved easily (although hackily) by simply accessing the current location via the global reference during lookup. Thanks to this addition to the `Ident.t` datatype and the changes in the lookup function, almost all the Eliom location mechanisms can be implemented without touching the typechecker's code itself.

Specialization Specialization is the action of taking a base type expression and bringing it to the current location. In the formalization, specialization only acts on binders, which is where location information is stored. Due to the fact that location is stored inside identifiers, the implementation is significantly more complex: we need to change the location annotations in the identifiers of the specialized expression. This is made

```
1 type t = {
2   stamp: int;
3   name: string;
4   mutable flags: int
5 }
```

Figure 6.4: The `Ident.t` datatype

		flag "c"	
		1	**0**
flag "s"	**1**	mixed	server
	0	client	base

Figure 6.5: Encoding of locations

even more delicate by the use of physical equality and mutations in the typechecker (the typechecker implements unification using a union-find algorithm embedded in the typedtree). The solution used for this purpose is to simulate copy of type expressions (as already done in the typechecker), except each identifier is copied with its flag corrected.

Binary compatibility Thanks to the various modifications made, the typedtree for Eliom is compatible with the OCaml one, but with some additional information embedded in PPX attributes and identifiers flags. In order to prevent the original typechecker to misuse this additional information, we use a different magic number for cmi files.

We hope that such binary compatibility will mean that the numerous OCaml tools, notably ocp-index, ocp-browser, odoc and merlin, can eventually be adapted to present Eliom's additional typing information.

6.5 Runtime and Serialization

We now present some practical consideration regarding the Eliom runtime. There are mainly two points of interest: the implementation of the Eliom primitives, Section 6.5.1, and the serialization format used for client-server communication, Section 6.5.2. We also discuss a small improvement to the slicing scheme in Section 6.5.3.

6.5.1 Primitives

The primitives used by the slicing scheme are presented formally in Section 5.1. The external signature for the primitives is shown in Figure 6.6. The main difference is that the notion of references has been further divided in `closure_id` and `inj_id` for fragment closure and injections references.

```
type closure_id
type inj_id

val%server fragment :
  closure_id -> 'injs -> 'a fragment
val%server push_fragments :
  id -> unit
val%server push_injection :
  inj_id -> 'a -> unit

val%client get_injection :
  inj_id -> 'a
val%client register_closure :
  closure_id -> ('a -> 'b) -> unit
val%client execute_fragments :
  id -> unit
```

Figure 6.6: API for Eliom primitives

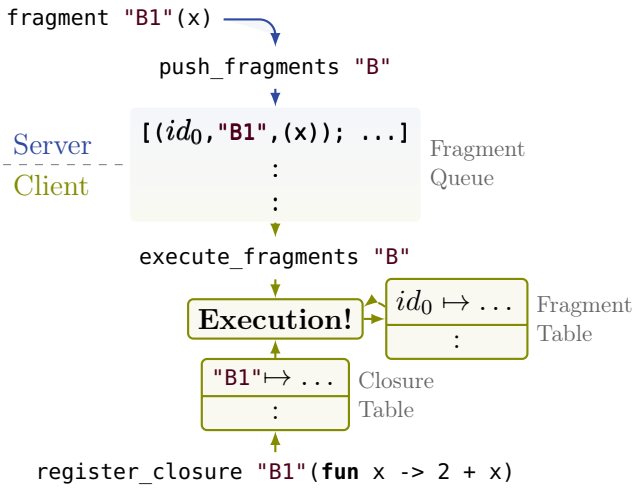


Figure 6.7: Execution of fragments

Transmitted data is represented by internal data structures which are shown in Figure 6.8. First, we use the type `poly` to (unsafely) represent arbitrary values. Similarly to closures and injections, fragments have their own kind of identifiers. Fragments on the server are represented by their identifier. The queue of fragments contains records of type `fragment`, which contains the unique identifier, the identifier of the associated closure and a tuple containing all the arguments. Injections are represented by the `injection` type, which is a pair of an identifier and the transmitted value. Finally, the transmitted information, which corresponds to the injection mapping ζ and the fragment queue ξ are represented by `global_data` and `request_data`.

The mapping of injection is represented as an array of id/values pairs. Instead of using an explicit end token in the queue of injection, we segment the queue for each section by using an array of array. The usage is very similar. Note that we use fairly basic data structures on purpose here, as they result in more compact serialized representations.

Global data is sent for all requests while request data is local to the request currently considered. More precisely, we can consider that the execution of a web server proceeds in two steps. First, we start the application: services are launched, database is accessed and so on. Fragments and injections executed during this start-up period are considered global. Then, the request handler is installed and start listening and responding to requests. Fragments executed during a request are of course local to it. This scheme allows to avoid recomputing things (global data is kept around and can be sent along the client program when needed) and also allows to avoid replaying some side effects. For example, the Eliom framework will reload only the minimum necessary when changing page inside a web application and thus only send new request data. Request data can only contain fragments (since injections are always global) and do not need end token (since everything is executed in one shot).

After executing the server program, we extract the resulting data, serialize it and add it to the generated HTML page. The client runtime will deserialize it, load it and use it to run the client program.

```
type poly (** Arbitrary values *)
val inject: 'a -> poly
val project: poly -> 'a

type fragment_id

type fragment = {
  closure : closure_id;
  args : poly;
  value : fragment_id
}

type injection = {
  id : inj_id;
  value : poly;
}

type global_data = {
  frags : fragment array array;
  injs : injection array;
}
type request_data = fragment array
```

Figure 6.8: Internal data structures for the Eliom runtime

6.5.2 Serialization format

Eliom typing and semantics are agnostic with regard to the serialization format, which means we can choose any of the numerous serialization methods available. Serialization methods usually are a compromise between several aspects: safety, composition, size and speed.

In our case, serialized messages are transmitted on the network, so we need a very compact serialization method. Furthermore, it should also be very fast, since serialization is one of the contention points of our system. Safety properties, on the other hand, are less important: type safety of serialization and deserialization is guaranteed by the Eliom type system. Messages could also be modified en-route. This is mitigated by two facts: HTTPS should always be used and, in case modifications are made, the worst outcome is a failure in the JavaScript client program, which is far less problematic than a crash in the server program. Finally, proper composition is not strictly needed. Indeed, Given our execution scheme for compiled program in Section 5.2, the complete arrays of injections can be serialized at once, after the execution of the server part of the program. The definition of converters, as discussed in Section 6.3, should be modular.

`Marshal` [marshal] is very fast and produces compact messages. The downside is that incompatible types at deserialization point might cause runtime errors. As detailed above, this is mitigated by Eliom's type system. Its distinct feature, compared to most other serialization formats, is that it preserves the sharing of OCaml values. This is extremely important to properly handle HTML across client-server boundaries (pointers to HTML elements on the server should point to the relevant DOM elements on the client). A safer and more modular alternative could be to use the "generic" library[5] by Balestrieri and Mauny [2016]. It leverages `Marshal` but provides improved safety checks by combining generic programming as popularized by approaches such as "Scrap Your Boilerplate" [Lämmel and Jones, 2003] with typed unmarshalling [Henry et al., 2012].

6.5.3 Optimized placement of sections

In Section 5.2, we present the slicing scheme for Eliom_ε programs. In particular, we present how to place *section* annotations `exec` () and `end` (). While this scheme is correct, it is also very wasteful.

Let us consider Figure 6.9. We see that the first `exec` is not necessary: all the fragments can be executed at once at the boundary between the set of server section and the following client section. Furthermore, since x is a value, it could not even lead to a fragment execution to begin with! While this has little impact on efficiency, the high number of sections can lead to issues in some browsers[6]. Additionally, it can produces bigger messages than necessary by introducing many `end` tokens.

Fortunately, this can be solved in a principled way with the following remarks:

- If a server declaration does not lead to any evaluation of client fragment, a `end`/`exec` pair is not needed.

[5] https://github.com/balez/generic
[6] Notably Safari. See https://github.com/ocsigen/eliom/pull/387 .

- In a succession of server declarations, only one pair of `end`/`exec` is needed, after the last successive server declaration.

The first remark can be approximated by looking if the declaration contains something that is not a value. If that is the case, we need a section, otherwise we do not. Given that OCaml and Eliom code mostly consist of function and type declarations, this is sufficient to eliminates more than two thirds of `end`/`exec` calls in medium-sized websites[6]. The second remark can easily be implemented by operating slicing only on the longest sequences of declarations with a common location.

```
let%server x = 1
let%server y =
  [%client ~%x + 1]

let%client res =
  ~%y
```

(a) Original Eliom code

```
execute_fragment "A"
register_closure "A1"
  (fun x -> x + 1)
execute_fragment "A"
let res =
  get_injection "A.z1"
```

(b) Naive client compilation

```

register_closure "A1"
  (fun x -> x + 1)
execute_fragment "A"
let res =
  get_injection "A.z1"
```

(c) Optimized version

Figure 6.9: Optimized placement of sections for a simple Eliom program

7 State of the art and comparison

> Togusa*: How great is the sum of thy thoughts? If I should count them, they are more in number than the sand.*
> Batou*: Psalms 139, Old Testament. The way you spout these spontaneous exotic references, I'd say your own external memory's pretty twisted.*
>
> Mamoru Oshii, *Ghost in the Shell 2: Innocence* (2004)

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 7.1, and the long lineage of ML programming languages, which we described in Section 3.4. One of the important contributions of Eliom is the use of a programming model similar to languages for distributed systems (Section 7.1.5) while using an execution model inspired by staged meta-programming (Section 7.2).

7.1 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.[1]

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[2] (mostly used for RPCs) and REST[3] (for Web APIs). A more recent attempt is the GraphQL [GraphQL] query language which

[1] 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
[2] https://en.wikipedia.org/wiki/SOAP
[3] https://en.wikipedia.org/wiki/Representational_state_transfer

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.

7.1.1 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 2.5. Such idioms are difficult to express when code locations are inferred. We demonstrate this with an example in the Ur/Web description.

- 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.

7.1.2 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.

7.1.3 Communications

Eliom uses asymmetric communication between client and server (see Section 2.2.2): 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 (Section 2.3.3), 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.

7.1.4 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,

2015a, page 10].[4] While the Eliom formalization is type safe, the Eliom implementation is not, due to the use of wrapping and Marshall (Section 6.3.2), which will fail at runtime on functional values.

Another remark is the distinction between client and server universes.[5] Eliom has separate type universes for client and server types (see Section 4.2.1). 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?

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.

7.1.5 Details on some specific approaches

We now provide an in-depth comparison with the most relevant approaches. A summary in Figure 7.1 classifies each approach according to the main distinctive features described in the previous paragraphs. Each language or framework is also described below.

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

[4] "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."

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

	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

Figure 7.1: 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.

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 2.5). For example, Example 7.1. 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. Other examples, such as the accordion widget (Section 2.7) are expressible only using reactive signals, which present a very different semantics.

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:

```
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>
      <xml><button value={name}
                   onclick={action}/>
      {buttons r}
     </xml>
 in
 return <xml>
   <body>{button_list l}</body>
 </xml>
end
```

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

Example 7.1: Programs building a list of buttons from a list of client side actions

∼-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 [Serrano and Queinnec, 2010, Boudol et al., 2012] and does present similarities to the interpreted Eliom semantics (Section 4.3). 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 7.2, 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.

```
links> fun adults(l) { for (x <- l) where (x.age >= 18) [(name = x.name)] } ;;
adults = fun : ([(age:Int,name:a|_)]) -> [(name:a)]

links> print ;;
print : (String) {wild}-> ()
```

Example 7.2: Small pieces of Links code

Meteor.js [Meteor.js] 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.

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.

Ekblad [2017] proposes an EDSL of Haskell 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 [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.

7.2 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, which we describe here.

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.

8 Conclusion

> *To the designer of programming languages, I say: unless you can support the paradigms I use when I program, or at least support my extending your language into one that does support my programming methods, I don't need your shiny new languages; [..] To persuade me of the merit of your language, you must show me how to construct programs in it.*
>
> Robert W. Floyd, *The paradigms of programming*

In this thesis, I presented the Eliom language, its design, formalization and implementation. Through Eliom, I also presented several related notions, such as ML languages, tierless web programming and staged meta-programming. At its core, Eliom combines the various insights made by staged meta-programming languages and the very powerful OCaml language in order to provide a safe and efficient programming language that allows to write client-server tierless programs in a convenient way.

One might note that, for most of this thesis, we do not talk about the Web all that much. Indeed, while Ocsigen is a Web programming framework, the language constructs we presented are not specific to Web programming. In fact, the minimal runtime developed during this thesis only needs the OCaml standard library to run, and can be compiled to any targets! Given the static nature of the Eliom programming language, the server and the client part are separated and compiled statically, so Eliom could be used to write more general client-server applications. Similarly, we believe that several techniques developed in the context of this thesis are of more general use. Notably, converters are a generic solution to make cross-stage persistence manipulable in a first class manner by programmers. From this perspective, Eliom can be seen as a general approach for type-safe client-server communications. Indeed Eliom does not force a specific programming style on the user. The additional primitives can be used for traditional Web programming techniques, as well as Model-View-Controller architectures or a more modern Functional Reactive approach. It simply turns out that combining those primitives with a programming language that supports modularity and encapsulation such as OCaml yields a powerful Web programming language.

Another focus of this thesis is to make Eliom usable in practice. Chapter 2 was dedicated to presenting the Eliom programming language from a user's perspective, with numerous examples demonstrating its usefulness for building Web applications and libraries. Chapter 6 presents how Eliom can be implemented as an actual extension of the OCaml compiler. In general, 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, hence the development of Eliom as an extension of OCaml. Since incremental compilation is

indispensable for any non-trivial programming projects, our design must be compatible with it. Given the existence of the OCaml module system, we needed good interaction between modularity, abstraction and the tierless annotations. All this led us to develop the various features presented in this thesis such as fragments, converters and location annotations inside modules.

Of course, there is still significant work to be done on the Eliom language. From a practical point of view, the new implementation needs to be improved and tested by more users. On the more theoretical aspects, mixed functors are still very much work in progress, as noted in Section 5.5. Instead of dwelling on these unfinished tasks, I would like, after three and a half years working on the subject, to offer my personal vision for the perfect tierless statically typed Web programming language.

My ideal tierless programming language Eliom is, as said above, very practically-minded. By extending OCaml, it leverages its many strengths but also inherits its design choices and weaknesses. As such, it represents a local optimum in the design space of tierless Web programming languages. I believe we can do better, and I will attempt here to give my ideas on how.

Before considering the tierless aspects, let me settle the kind of language I will consider here. As should be apparent by now, I strongly prefer statically-typed strict-by-default languages. Dynamic languages do have numerous advantages, for example in the context of distributed systems like Erlang or in very dynamic settings where the Scheme family excels. Indeed, Hop already provides a very convincing solution for dynamically-typed tierless Web programming. My personal taste[1], however, goes to static typing, both for the numerous benefits it provides in term of soundness and convenience, but also because the discipline it offers help me organize my own thoughts on how and what to do while exploring the complicated maze of design decisions that is programming.

Given this context, I believe that the modularity and encapsulation properties of a strong module system with abstract datatypes are one of the best programming tools available in modern programming languages. Modules give the programmer the ability to manipulate structured programs directly inside the language and allows to enforce invariants inside module boundaries, as demonstrated on multiple occasions in Chapter 2. These capabilities are essential for any kind of modular programming, including Web programming. A module system providing these capabilities can take many forms: the "classical" ML module system, the more practical OCaml one; more experimental module systems such as 1ML [Rossberg et al., 2014] or even something quite different, such as Backpack [Kilpatrick et al., 2014]. All these module systems provide the ability to hide internal details through abstraction and to manipulate modules in very powerful ways, thus providing the necessary tools for modular large scale programming.

On the tierless aspects, syntactic annotations are essential to provide finer control, as argued in several occasions in this thesis. However, I believe a type-and-effect system in the style of Eff [Bauer and Pretnar, 2012] could provide a tighter integration between

[1]For the willing, I can propose other lively debates: Vim vs. Emacs; Zelazny vs. Sanderson, K. Dick vs. Asimov; Tomatoes vs. Potatoes.

the type system and the tierless annotations. Indeed, we would have two effects: "client" and "server". Fragments would only contain code that has no "server" effect, but would produce a server effect themselves. This could allow to elide some of the most obvious annotations, hence allowing a programming style similar to Ur/Web's implicitness while still providing the necessary control when needed. Furthermore, location information would be exposed at the module level directly through the effects systems, thus replacing the need for annotations in module types. Effects would then play a similar role than modalities in ML5. Links already demonstrated that such an effect-based language can be compiled down to quotations [Cheney et al., 2014], which would allow to keep the efficient execution model of Eliom.

Finally, while there is a rather large tooling ecosystem surrounding Web languages, the same cannot be said about tierless languages (except maybe Hop.js). In particular, a REPL for a statically sliced tierless language would both be technically challenging and provide a more exploratory style of programming.

This design can be decomposed in a combination of powerful general-purpose features (here, modularity, abstraction and effect systems) and a few selected specific-purpose primitives (our two new effects). This combination allows to build up new idioms for a chosen domain and provides good integration into existing languages. By exposing orthogonal features that interact well instead of baking complex constructions in the language, we can provide tools that are more flexible and easier to extend by programmers. I believe this is a nice way of designing programming languages, and I hope to pursue it in other domains in the future.

Bibliography

N. Amin and T. Rompf. Type soundness proofs with definitional interpreters. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 666–679. ACM, 2017. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837. URL http://dl.acm.org/citation.cfm?id=3009866.

B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In G. C. Necula and P. Wadler, editors, *Proceeding of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 3–15. ACM, 2008. URL http://arthur.chargueraud.org/research/2007/binders/.

V. Balat. Rethinking Traditional Web Interaction. In *Proceedings of the Eighth International Conference on Internet and Web Applications and Services*, Rome, Italy, 2013.

V. Balat. Rethinking traditional web interaction: Theory and implementation. *International Journal on Advances in Internet Technology*, 2014. ISSN 1942-2652. URL http://www.iariajournals.org/internet_technology/.

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

F. Balestrieri and M. Mauny. Generic programming in OCaml. *OCaml Workshop*, 2016.

A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *CoRR*, abs/1203.1539, 2012. URL http://arxiv.org/abs/1203.1539.

A. Bawden. Quasiquotation in lisp. In *PEPM*, pages 4–12. University of Aarhus, 1999.

G. Boudol, Z. Luo, T. Rezk, and M. Serrano. Reasoning about Web applications: An operational semantics for HOP. *ACM Trans. Program. Lang. Syst.*, 34(2):10, 2012.

J. Cheney, S. Lindley, G. Radanne, and P. Wadler. Effective quotation: relating approaches to language-integrated query. In W. Chin and J. Hage, editors, *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation, PEPM 2014, January 20-21, 2014, San Diego, California, USA*, pages 15–26. ACM, 2014. doi: 10.1145/2543728.2543738. URL http://doi.acm.org/10.1145/2543728.2543738.

A. Chlipala. 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, pages 153–165, New York, NY, USA, 2015a. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677004. URL http://doi.acm.org/10.1145/2676726.2677004.

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

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

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

D. Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, CMU, May 2005. URL https://people.mpi-sws.org/~dreyer/thesis/main.pdf.

A. Ekblad. A meta-edsl for distributed web applications. In I. S. Diatchki, editor, *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 75–85. ACM, 2017. doi: 10.1145/3122955.3122969. URL http://doi.acm.org/10.1145/3122955.3122969.

A. Ekblad and K. Claessen. A seamless, client-centric programming model for type safe web applications. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 79–89, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3041-1. doi: 10.1145/2633357.2633367. URL http://doi.acm.org/10.1145/2633357.2633367.

Eliom. *Eliom web site*. https://ocsigen.org/eliom, 2017.

Immutable. *Immutable*. Facebook, https://facebook.github.io/immutable-js/, 2017.

Reactjs. *Reactjs*. Facebook, https://reactjs.org/, 2017.

N. Feltman, C. Angiuli, U. A. Acar, and K. Fatahalian. Automatically splitting a two-stage lambda calculus. In Thiemann [2016], pages 255–281. ISBN 978-3-662-49497-4. doi: 10.1007/978-3-662-49498-1_11. URL http://dx.doi.org/10.1007/978-3-662-49498-1_11.

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

GraphQL. Graphql, 2016. URL http://graphql.org/.

G. Henry, M. Mauny, E. Chailloux, and P. Manoury. Typing unmarshalling without marshalling types. In P. Thiemann and R. B. Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 287–298. ACM, 2012. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364569. URL http://doi.acm.org/10.1145/2364527.2364569.

J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989. doi: 10.1093/comjnl/32.2.98. URL https://doi.org/10.1093/comjnl/32.2.98.

S. Kilpatrick, D. Dreyer, S. L. P. Jones, and S. Marlow. Backpack: retrofitting haskell with interfaces. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 19–32. ACM, 2014. doi: 10.1145/2535838.2535884. URL http://doi.acm.org/10.1145/2535838.2535884.

O. Kiselyov. The design and implementation of BER metaocaml - system description. In M. Codish and E. Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 86–102. Springer, 2014. ISBN 978-3-319-07150-3. doi: 10.1007/978-3-319-07151-0_6. URL http://dx.doi.org/10.1007/978-3-319-07151-0_6.

O. Kiselyov. Generating code with polymorphic let: A ballad of value restriction, copying and sharing. In J. Yallop and D. Doligez, editors, *Proceedings ML Family / OCaml Users and Developers workshops, ML Family/OCaml 2015, Vancouver, Canada, 3rd & 4th September 2015.*, volume 241 of *EPTCS*, pages 1–22, 2015. doi: 10.4204/EPTCS.241.1. URL https://doi.org/10.4204/EPTCS.241.1.

O. Kiselyov, A. Biboudis, N. Palladinos, and Y. Smaragdakis. Stream fusion, to completeness. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 285–299. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3009880.

R. Lämmel and S. L. P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. In Z. Shao and P. Lee, editors, *Proceedings of TLDI'03: 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, New Orleans, Louisiana, USA, January 18, 2003*, pages 26–37. ACM, 2003. ISBN 1-58113-649-8. doi: 10.1145/604174.604179. URL http://doi.acm.org/10.1145/604174.604179.

D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of standard ML. In M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 173–184. ACM, 2007. ISBN 1-59593-575-4. doi: 10.1145/1190216.1190245. URL http://doi.acm.org/10.1145/1190216.1190245.

X. Leroy. Manifest types, modules, and separate compilation. In H. Boehm, B. Lang, and D. M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 109–122. ACM Press, 1994. doi: 10.1145/174675.176926. URL http://doi.acm.org/10.1145/174675.176926.

X. Leroy. Applicative functors and fully transparent higher-order modules. In R. K. Cytron and P. Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 142–153. ACM Press, 1995. doi: 10.1145/199448.199476. URL http://doi.acm.org/10.1145/199448.199476.

X. Leroy. A syntactic theory of type generativity and sharing. *J. Funct. Program.*, 6 (5):667–698, 1996. doi: 10.1017/S0956796800001933. URL http://dx.doi.org/10.1017/S0956796800001933.

X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.04, Documentation and user's manual*. Projet Gallium, INRIA, Nov. 2016.

S. Lindley and J. Cheney. Row-based effect types for database integration. In B. C. Pierce, editor, *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 91–102. ACM, 2012. doi: 10.1145/2103786.2103798. URL http://doi.acm.org/10.1145/2103786.2103798.

F. Loitsch and M. Serrano. Hop client-side compilation. In M. T. Morazán, editor, *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007.*, volume 8 of *Trends in Functional Programming*, pages 141–158. Intellect, 2007.

D. B. MacQueen. Modules for standard ML. In *LISP and Functional Programming*, pages 198–207, 1984.

marshal. *OCaml Standard Library – Marshal*. http://caml.inria.fr/pub/docs/manual-ocaml/libref/Marshal.html, 2016.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3(4):184–195, 1960. doi: 10.1145/367177.367199. URL http://doi.acm.org/10.1145/367177.367199.

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

R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17 (3):348–375, 1978. doi: 10.1016/0022-0000(78)90014-4. URL http://dx.doi.org/10.1016/0022-0000(78)90014-4.

R. Milner, M. Tofte, and R. Harper. *Definition of standard ML*. MIT Press, 1990. ISBN 978-0-262-63132-7.

Y. Minsky, A. Madhavapeddy, and J. Hickey. *Real World OCaml - Functional Programming for the Masses*. O'Reilly, 2013. ISBN 978-1-4493-2391-2. URL https://realworldocaml.org/.

K. Nakata and T. Uustalu. Trace-based coinductive operational semantics for while. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 375–390. Springer, 2009. doi: 10.1007/978-3-642-03359-9_26. URL https://doi.org/10.1007/978-3-642-03359-9_26.

O. Nicole. Bringing typed, modular macros to ocaml, 2016. URL https://oliviernicole.github.io/about_macros.html.

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

S. Owens. A sound semantics for ocamllight. In S. Drossopoulou, editor, *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*, volume 4960 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2008. doi: 10.1007/978-3-540-78739-6_1. URL https://doi.org/10.1007/978-3-540-78739-6_1.

S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan. Functional big-step semantics. In Thiemann [2016], pages 589–615. ISBN 978-3-662-49497-4. doi: 10.1007/978-3-662-49498-1_23. URL https://doi.org/10.1007/978-3-662-49498-1_23.

L. Philips, C. De Roover, T. Van Cutsem, and W. De Meuter. 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, pages 69–81, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3210-1. doi: 10.1145/2661136.2661146. URL http://doi.acm.org/10.1145/2661136.2661146.

L. Philips, J. D. Koster, W. D. Meuter, and C. D. Roover. Dependence-driven delimited CPS transformation for javascript. In B. Fischer and I. Schaefer, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 59–69. ACM, 2016. doi: 10.1145/2993236.2993243. URL http://doi.acm.org/10.1145/2993236.2993243.

C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In M. Odersky and P. Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada,*

September 18-21, 2000., pages 23–33. ACM, 2000. doi: 10.1145/351240.351243. URL http://doi.acm.org/10.1145/351240.351243.

C. Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Notices*, 38(2):57–64, 2003. doi: 10.1145/772970.772977. URL http://doi.acm.org/10.1145/772970.772977.

C. Queinnec. Continuations and web servers. *Higher-Order and Symbolic Computation*, 17(4):277–295, 2004. doi: 10.1007/s10990-004-4866-z. URL https://doi.org/10.1007/s10990-004-4866-z.

React. *React*. http://erratique.ch/software/react, 2017.

A. Rossberg, C. V. Russo, and D. Dreyer. F-ing modules. *J. Funct. Program.*, 24(5): 529–607, 2014. doi: 10.1017/S0956796814000264. URL https://doi.org/10.1017/S0956796814000264.

G. Scherer and J. Vouillon. Macaque : Interrogation sûre et flexible de base de données depuis OCaml. In *Ving et unième journées francophones des langages applicatifs*, Studia Informatica Universalis, pages –, La Ciotat, France, Jan. 2010. Hermann. URL https://hal.archives-ouvertes.fr/hal-00495977.

M. Serrano and V. Prunet. A glimpse of hopjs. In J. Garrigue, G. Keller, and E. Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 180–192. ACM, 2016. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951916. URL http://doi.acm.org/10.1145/2951913.2951916.

M. Serrano and C. Queinnec. A multi-tier semantics for Hop. *Higher-Order and Symbolic Computation*, 23(4):409–431, 2010.

M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the Web 2.0. In *OOPSLA Companion*, pages 975–985, 2006.

P. Sewell, J. J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. *J. Funct. Program.*, 17(4-5):547–612, 2007. doi: 10.1017/S0956796807006442. URL http://dx.doi.org/10.1017/S0956796807006442.

Shared reactive programming. *Shared React*. https://ocsigen.org/eliom/5.0/manual/clientserver-react, 2017.

T. Sheard and S. L. P. Jones. Template meta-programming for haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

J. Siek. Type safety in three easy lemmas, May 2013. URL https://siek.blogspot.fr/2013/05/type-safety-in-three-easy-lemmas.html.

D. Swasey, T. M. VII, K. Crary, and R. Harper. A separate compilation extension to standard ML. In A. Kennedy and F. Pottier, editors, *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, pages 32–42. ACM, 2006. doi: 10.1145/1159876.1159883. URL http://doi.acm.org/10.1145/1159876.1159883.

P. Thiemann, editor. *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*, volume 9632 of *Lecture Notes in Computer Science*, 2016. Springer. ISBN 978-3-662-49497-4. doi: 10.1007/978-3-662-49498-1. URL http://dx.doi.org/10.1007/978-3-662-49498-1.

M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1988.

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

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

T. M. VII, K. Crary, and R. Harper. Type-safe distributed programming with ML5. In G. Barthe and C. Fournet, editors, *TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 108–123. Springer, 2007. ISBN 978-3-540-78662-7.

J. Vouillon. Lwt: a cooperative thread library. In E. Sumii, editor, *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, pages 3–12. ACM, 2008. ISBN 978-1-60558-062-3. doi: 10.1145/1411304.1411307. URL http://doi.acm.org/10.1145/1411304.1411307.

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

P. Wadler. Theorems for free! In J. E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989. doi: 10.1145/99370.99404. URL http://doi.acm.org/10.1145/99370.99404.

L. White, F. Bour, and J. Yallop. Modular implicits. In O. Kiselyov and J. Garrigue, editors, *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014.*, volume 198 of *EPTCS*, pages 22–63, 2014. doi: 10.4204/EPTCS.198.2. URL http://dx.doi.org/10.4204/EPTCS.198.2.

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

J. Yallop and L. White. Modular macros. *OCaml Workshop*, 2015. URL http://www.lpw25.net/ocaml2015-abs1.pdf.