Type-safe multilanguage programming

Computer Science Tripos Part II
Magdalene College
University of Cambridge
May 15, 2013
Proforma

Name: Eduardo Munoz
College: Magdalene College
Project Title: Type-safe multilanguage programming
Examination: Part II Computer Science, June 2013
Word Count: 11,982
Project Originator: Eduardo Munoz
Supervisor: Tomas Petricek

Original aims of the project

The aim of the project was to implement a type-safe library to provide interoperability between the F# and JavaScript programming languages, using the theory in J.B. Matthews’ Ph.D. dissertation [1]. Matthews presents the semantics for the lump and natural embeddings, and the aim is to understand and implement this theory.

Work completed

Both embeddings have been implemented for F# and JavaScript, rendering a type-safe system that cannot produce a runtime error in F# due to an error in the glue code between the languages. The resulting library, MiXture, was systematically tested following industry standards and quantitatively and qualitatively evaluated in order to ensure the specifications have been met. The natural embedding has been extended, and the added cases for the proof of type-safety are provided in Appendix A.

Special difficulties

None.
Declaration

I, Eduardo Munoz of Magdalene College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I give permission for my dissertation to be made available in the archive area of the Laboratory’s website.

Signed

Date
# Contents

1 Introduction ........................................................................................................... 1
   1.1 Motivation .................................................................................................... 1
   1.2 Current multilanguage systems ................................................................. 2
      1.2.1 Foreign function interfaces ................................................................. 2
      1.2.2 Multilanguage runtimes ..................................................................... 2
      1.2.3 Embedded interpreters ....................................................................... 3
   1.3 Difficulties ................................................................................................... 4
      1.3.1 Type system ....................................................................................... 4
      1.3.2 Values ............................................................................................... 5
      1.3.3 Evaluation strategy .......................................................................... 5
   1.4 Project aims ................................................................................................. 6
   1.5 Work completed .......................................................................................... 6

2 Preparation ............................................................................................................ 7
   2.1 Embeddings ............................................................................................... 7
      2.1.1 Syntactic kinds of embedding ............................................................ 7
      2.1.2 Base calculi for describing the embeddings ....................................... 8
      2.1.3 Lump embedding ............................................................................. 9
      2.1.4 Natural embedding .......................................................................... 10
      2.1.5 Type-indexed embedding and projection algorithms ....................... 12
   2.2 V8 JavaScript engine .................................................................................... 13
   2.3 Software engineering techniques ............................................................... 14
      2.3.1 Development methodology ............................................................... 15
      2.3.2 Requirements analysis ..................................................................... 15
      2.3.3 Practical preparation ........................................................................ 17
      2.3.4 Development tools ........................................................................... 17
   2.4 Summary .................................................................................................... 18

3 Implementation ...................................................................................................... 19
   3.1 High-level code structure .......................................................................... 19
   3.2 Lump embedding ....................................................................................... 20
      3.2.1 Implementation of lumps .................................................................. 20
      3.2.2 Cross-language communication ...................................................... 20
      3.2.3 Memory management ...................................................................... 23
   3.3 Natural embedding ..................................................................................... 24
      3.3.1 Representation of a JavaScript handle in F# ................................... 24
      3.3.2 Embedding/projection pairs ............................................................... 24
      3.3.3 Notation ............................................................................................ 26
# List of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The design space of type systems and the location of some languages within it, adapted from [6].</td>
</tr>
<tr>
<td>2.3</td>
<td>Toy calculi.</td>
</tr>
<tr>
<td>2.4</td>
<td>Extensions to calculi $\mathcal{F}$ and $\mathcal{J}$ (Figure 2.3) to form the lump embedding.</td>
</tr>
<tr>
<td>2.5</td>
<td>Example of the lump embedding for $\mathcal{F}$ (host) and $\mathcal{J}$ (guest).</td>
</tr>
<tr>
<td>2.6</td>
<td>Extensions to languages $\mathcal{F}$ and $\mathcal{J}$ (Figure 2.3) to form the natural embedding.</td>
</tr>
<tr>
<td>2.7</td>
<td>V8 handles overview, from the V8 documentation [13].</td>
</tr>
<tr>
<td>2.8</td>
<td>Gantt chart illustrating the project schedule and its completion as of January 18, 2013.</td>
</tr>
<tr>
<td>2.9</td>
<td>UML use case diagram for the interaction of a developer and the multilanguage system.</td>
</tr>
<tr>
<td>3.1</td>
<td>High-level design of MIXture.</td>
</tr>
<tr>
<td>3.2</td>
<td>Interoperation between F#, V8 and JavaScript for the lump embedding.</td>
</tr>
<tr>
<td>3.3</td>
<td>UML class diagram for lumps.</td>
</tr>
<tr>
<td>3.4</td>
<td>Translation rules for JavaScript Number. The subscript in the numeric values indicates the programming language they belong to, where JS stands for JavaScript. truncate sets to zero the decimal digits of a floating point number: e.g., truncate(3.14) = 3; round maps a non-negative floating point number $f$ to $\text{floor}(f)$: e.g., round(3.14) = 3; and a non-positive number $f$ to $\text{ceiling}(f)$: e.g., round(-3.14) = -3, where $\text{floor}$ and $\text{ceiling}$ map floating point numbers to the smallest following integer and the largest previous integer, respectively. Error stands for raising a ProjectionException.</td>
</tr>
<tr>
<td>3.5</td>
<td>Embed and project rules for functions.</td>
</tr>
<tr>
<td>3.6</td>
<td>Extensions to Figure 2.6 to include lists.</td>
</tr>
<tr>
<td>3.7</td>
<td>Project rule for F# lists.</td>
</tr>
<tr>
<td>3.8</td>
<td>Project rule for F# records.</td>
</tr>
<tr>
<td>3.9</td>
<td>Memory management for embedded functions. The labels on top of the lines denote the embedding steps (in brackets) and go in the direction of the filled arrow heads. The labels under the lines denote the memory management process and go in the direction of empty arrow heads.</td>
</tr>
<tr>
<td>4.1</td>
<td>ECP tests run in MonoDevelop.</td>
</tr>
<tr>
<td>4.2</td>
<td>Heapshot analysis results, sample number in parenthesis.</td>
</tr>
<tr>
<td>4.3</td>
<td>Screenshot of some of the performance tests being run. Each dot “.” represents one run of a test: in this case, each test was run 10 times.</td>
</tr>
<tr>
<td>4.4</td>
<td>Benchmarking CPU execution time of monolingual and multilanguage systems. The input for each test is specified in brackets after each name.</td>
</tr>
</tbody>
</table>
4.5 Difference of means of CPU execution times using MiXture and native run-
times. ................................................................. 51
4.6 Performance tests for project/ embed for different data types. .... 52
4.7 Benchmarking CPU execution time versus the length of the string being
embedded (blue) and projected (red). ............................. 53
4.8 Benchmarking CPU execution time versus the record recursion level 55
List of code listings

1.1 Example of embedding the OCaml function hypot into Lua using Lua-ML, adapted from [5]. .......................................................... 3
1.2 Embedding a polymorphic F#-defined recursive function into JavaScript. . 6
2.1 bool pair from Lua-ML (lines 407–408 from the source file luavalue.nw [12]). 12
3.1 Computer vision example of foreignApply, where an F# function is wrapped
in a (int list -> int list) FSLump and is applied to an image I and a
convolution kernel K. The types of the variables are given in an ML-like
specification in a comment, as recommended by Felleisen [21]. ............... 21
3.2 Definition of the record type (’a, ’b) ep, an embedding/projection pair
used to translate values between F# and JavaScript. ......................... 24
3.3 Illustrating the use of active patterns to pattern match a JavaScript handle.
Note that not all active patterns have been included. ......................... 26
3.4 Projecting a ternary JavaScript function into F#, creating a curried function. 30
3.5 Interactive sessions transcript showing that naively embedding F# polymor-
phic functions is not type-safe. .................................................. 38
3.6 Obtaining the polymorphic type information of Array.append and fst. 38
3.7 Interactive sessions transcript illustrating that using embed_poly_func is
type-safe for F# polymorphic functions. Compare with Listing 3.5 ......... 39
3.8 Projecting a JavaScript function into a polymorphic F# function. ........ 40
C.1 Embedding a resolution function into JavaScript and an interactive session
transcript. .............................................................................. 78
C.2 Embedding a resolution function into Lua and an interactive session transcript. 79
Chapter 1

Introduction

This dissertation describes the development of a library that allows programs to be written in the two programming languages F# and JavaScript, in a type-safe manner, and with a great level of integration between the languages.

In this chapter, we present an overview on the topic of multilanguage systems: how practical they are, what challenges arise when implementing them, and some existing solutions.

1.1 Motivation

Selecting the right tools when developing a large scale software system is crucial for the success of the project. For this reason, most modern systems are developed in several programming languages, including scripting, compiled, functional, imperative, domain specific, etc. However, the interaction between these different components may be problematic. For instance, the calling conventions of one language might not correspond to those of another, hence causing incompatibilities.

A multilanguage system allows a programmer to use two or more languages by providing ways of cross-language communication. The host language is the default runtime environment and provides the glue code for multilanguage programs, while the guest language is the foreign language in the host environment. There are several degrees in which languages interoperate: from coarse-grained interoperability (only call void procedures from other languages) to finer-grained levels (make all values defined in one language available in the other one).
1.2 Current multilanguage systems

There are several approaches to implement multilanguage systems, each having different advantages and disadvantages. They are outlined in the following sections.

1.2.1 Foreign function interfaces

Foreign function interfaces (FFI) allow programs written in one language to call routines in another. This mechanism is widespread and is usually implemented for pairs of languages, typically one of them being high-level and the other one low-level (commonly C). The interoperability is often bidirectional (the host language can invoke guest callables and vice versa), although calling the guest language is usually more convenient.

Most programming languages have an FFI with native code. Some examples include:

- The Java Native Interface allows Java code to incorporate native code written in languages such as C and C++.
- ctypes is a foreign function library for Python and C.

It may be necessary to unify the rules of the two language specifications, e.g., memory management, calling conventions, etc. Tracking global invariants is also problematic, which is needed for garbage collection if one of the languages implements it. Furthermore, the task of writing the glue code between the languages is usually excessively verbose, which many tools try to simplify.

1.2.2 Multilanguage runtimes

In these systems, several programming languages target the same runtime architecture (e.g., virtual machine), allowing a richer interaction between the languages, such as inheriting classes in one language defined in another.

Two multilanguage runtime systems have received special attention in the last two decades: Sun’s Java Virtual Machine and Microsoft’s Common Language Runtime. They are both targeted by several languages, including:

- JVM: Java, Scala, Clojure (Lisp), Jython, JRuby, etc.;
- CLR: Microsoft’s languages C#, F#, VB, as well as IronPython, IronRuby, etc.
1.2. CURRENT MULTILANGUAGE SYSTEMS

The main advantage of this mechanism of language interoperability is the standardization of system-level services [1]. However, the common core must provide common services to potentially different languages, which is considered to be overcome reasonably well for the CLR with C# and F# [2].

1.2.3 Embedded interpreters

This approach consists of implementing an interpreter of the guest language in the host language, where translation of values is performed by embedding and projection algorithms [3]. A type-indexed embedding/projection pair is a type-specific value that allows embedding (from host to guest translation) and projection (from guest to host translation) of values. Note that, in the literature, to embed a value is sometimes called to lift or wrap a value, and to project a value is to unwrap a value.

Lua-ML [4, 5] is an example of this technique, where a Lua interpreter is implemented in OCaml. In Listing 1.1 we can see how a function implemented in OCaml, hypot, is made available to be used by the Lua interpreter. In line 2, an embed/project pair is created for the type float -> float -> float, and is then applied to hypot. The purpose of the non-standard **-> and result operators of Lua-ML is briefly discussed in §3.3.5.

```ocaml
1 let my_hypot =
2  let f = func (float **-> float **-> result float)
3  in f.embed hypot
```

Listing 1.1: Example of embedding the OCaml function hypot into Lua using Lua-ML, adapted from [5].

The main advantage of embedded interpreters is the ease with which guest values can be exposed to the host and vice versa. However, there are drawbacks to this technique, such as a) it generally requires one to develop a new interpreter for the purpose of interoperability only, and b) the asymmetry between the guest and host.

A related approach is used in this project, where we embed a JavaScript engine in F#. We then use embedding and projection algorithms to provide a cross-communication mechanism between the languages for a number of different types, striving not to weaken type-safety in F#.
1.3 Difficulties

There are some factors that make achieving full language interoperability a hard task. Specifically, programming languages can differ in three axes: the type system, the representable values (e.g., different numerical values), and the evaluation strategy. Consequently, smoothing the transition between the two languages requires specific solutions for each axis. The main concern in this dissertation is differing type systems and values.

1.3.1 Type system

Type systems are a formal method to help ensure a system behaves correctly, and they can vary in a number of dimensions, including dynamic vs static and strong vs weak. This is illustrated in Figure 1.1.

![Figure 1.1: The design space of type systems and the location of some languages within it, adapted from [6].](image)

Multilanguage systems of a very weakly typed language and a strongly typed one threaten type safety of the resulting implementation. They raise the following questions:

- How to assign a type for a value in the untyped language when being translated into the typed language?
- How to ensure that the untyped language will not use foreign values from the strongly typed language in a non-safe manner?
1.3. DIFFICULTIES

1.3.2 Values

Some values clearly correspond to others in different languages. For instance, strings in most languages are a sequence of char values representing some sort of text. But even with these “corresponding” values, the internal byte representation can differ and have some subtleties (e.g., strings in Java are immutable, while they are mutable in C).

Moreover, the set of values expressible in a language does not necessarily match that in another. For example, object values in JavaScript correspond to a certain extent to F# records, since both are key-value collections. However, JavaScript objects support prototype-based inheritance, but F# records don’t support inheritance at all.

For this reason, the value conversions between F# and JavaScript is not type-directed, that is, the single type of the value does not uniquely characterize the conversion. Rather, a type “strategy” is required in order to specify the conversion to be performed; we say conversions are type-mapped. An example of this is the type of a JavaScript object being mapped to multiple F# record types.

1.3.3 Evaluation strategy

There are two main evaluation strategies in programming languages in use today: strict and non-strict evaluation. Strict evaluation reduces terms from the innermost brackets and works outwards, whereas non-strict evaluation proceeds from the outside inwards. As a result, in non-strict languages, function applications can have a definite value although an argument is undefined, because some sub-expressions might be eliminated by outer reductions. The most popular implementations of these systems are eager evaluation (strict), in which all arguments are evaluated before performing a function application; and lazy evaluation (non-strict), in which the arguments to a function are not evaluated before the function is called, and stored for subsequent uses.

Rudiak-Gould et al. present a calculus to serve as an intermediate language capable of embedding ML-like (strict and eager) and Haskell-like (non-strict and lazy) languages, as well as compiling them efficiently.

This dissertation does not address this axis for two reasons. First, both languages used in this work (F# and JavaScript) evaluate eagerly (so there are no differences); and second, mixed strict/non-strict programming is an open research problem, out of the scope of this project.

---

1 It could also be argued that the discussion is about types (but not type systems). Types are seen as sets of values, so this discussion applies both to “corresponding” types and values.
1.4 Project aims

The aim of this project is to design and implement a multilanguage system for JavaScript and F#. JavaScript is treated as a general purpose language in this dissertation and not as a client-side language in a browser. These two languages present an interesting combination: they both support the functional programming paradigm (so functions are values to be translated), but JavaScript is mainly untyped, while F# is strongly and statically typed with type inference. This presents some challenges that have been overcome in this project.

The system implemented consists of two types of interoperability between languages, described in J.B. Matthews’ Ph.D. dissertation [1], as well as other research papers [8][9]: the bump and the natural embeddings. The implementation of the natural embedding is based on the embedding interpreters approach [3][4][5], with some original additions.

1.5 Work completed

This project involved studying research papers, as well as producing substantial pieces of software: 1,500 lines only of F# (benefits of functional programming), and 500 of C++.

The system implemented for this project allows source code as illustrated in the F# interactive session transcript in Listing 1.2. Here, the recursive and polymorphic function List.append is embedded into JavaScript in the form of jappend (line 3), and is registered in a JavaScript context (line 5). JavaScript source code is then executed: jappend is applied to two lists of Numbers and two lists of strings.

```
CHAPTER 1. INTRODUCTION

1.4 Project aims

The aim of this project is to design and implement a multilanguage system for JavaScript and F#. JavaScript is treated as a general purpose language in this dissertation and not as a client-side language in a browser. These two languages present an interesting combination: they both support the functional programming paradigm (so functions are values to be translated), but JavaScript is mainly untyped, while F# is strongly and statically typed with type inference. This presents some challenges that have been overcome in this project.

The system implemented consists of two types of interoperability between languages, described in J.B. Matthews’ Ph.D. dissertation [1], as well as other research papers [8][9]: the bump and the natural embeddings. The implementation of the natural embedding is based on the embedding interpreters approach [3][4][5], with some original additions.

1.5 Work completed

This project involved studying research papers, as well as producing substantial pieces of software: 1,500 lines only of F# (benefits of functional programming), and 500 of C++.

The system implemented for this project allows source code as illustrated in the F# interactive session transcript in Listing 1.2. Here, the recursive and polymorphic function List.append is embedded into JavaScript in the form of jappend (line 3), and is registered in a JavaScript context (line 5). JavaScript source code is then executed: jappend is applied to two lists of Numbers and two lists of strings.

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Chapter 2

Preparation

This chapter outlines the research carried out before implementing the system of this project. Specifically, we describe the lump and natural embeddings (§2.1.3, §2.1.4), and the use of type-indexed embedding and projection algorithms as an implementation of the natural embedding (§2.1.5) [3, 5]. Finally, we provide an introduction to the engine used to manage JavaScript code (§2.2) and the software engineering techniques employed (§2.3).

2.1 Embeddings

This section introduces two “toy” calculi — F and J — in order to introduce the concepts of the lump and natural embeddings. In the implementation chapter, calculus F stands for F# and J for JavaScript. These calculi are particularly simple in order to introduce the core concepts rather than dwelling on the details of the operational semantics. The limitations of these embeddings are explored in §2.1.3.1 and §2.1.4.1, and an implementation strategy of the natural embedding is discussed in §2.1.5.

2.1.1 Syntactic kinds of embedding

There exists a range of different ways in which the syntax of one language may be embedded into another. One approach consists in combining the abstract syntax of both languages by the use of language boundaries. This is the approach taken by Matthews to describe the operational semantics of multilanguage programs, also used throughout this section. Another method of embedding a language is to produce strings of the guest language in the host language. This is the strategy followed in existing implementations of this theory [3, 5], and in this project, due to the fact that JavaScript is not a compiled language and the use of a JavaScript engine makes it easy to embed it as a string in F#. Nevertheless, this does not threaten type-safety in F#, as we will discuss in later sections.
2.1.2 Base calculi for describing the embeddings

The grammars and operational semantics of two very simple calculi are given in Figure 2.3, which will be used to explain the lump and natural embeddings. Observe that we use Felleisen-style context-sensitive reduction semantics to specify the operational semantics.

\( \mathcal{F} \) is specified with terms in a green bold serif font (remember, the color “Forest green” serif \( \mathcal{F} \) for the \( \mathcal{F} \) language), and \( \mathcal{J} \), in a blue sans-serif font. For instance, \( e \) denotes an \( \mathcal{F} \) expression, whereas \( e \) is a \( \mathcal{J} \) expression. This will help in distinguishing the language a term belongs to when the syntax of both languages is allowed in a single expression.

We can see that both calculi have a similar syntax and differ only in their type systems. Language \( \mathcal{F} \) is strictly typed, whereas language \( \mathcal{J} \) is untyped (all expressions have the “JavaScript Type” JT). These calculi are restricted to function values and booleans only (true and false are the syntactic terms in language \( \mathcal{F} \) for the boolean values true and false, and similarly for true and false), as their sole purpose is to illustrate how the embeddings work. These grammars will be augmented in both kinds of embedding to allow interoperability between \( \mathcal{F} \) and \( \mathcal{J} \).

\[
\begin{align*}
e &::= x \mid v \mid (e \; e) \\
\mathcal{E} &::= [\;]_{\mathcal{F}} \mid (\mathcal{E} \; e) \mid (v \; \mathcal{E}) \\
\tau &::= \text{bool} \mid \tau \rightarrow \tau
\end{align*}
\]

\( \text{(BOOL)} \quad \Gamma \vdash b : \text{bool} \), if \( b \in \{\text{true, false}\} \)

\( \text{(FN}_F \text{)} \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \quad \frac{}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \rightarrow \tau_2} \)

\( \mathcal{E}[(\lambda x : \tau . e) \; v] \rightarrow \mathcal{E}[e \; [v/x]] \)

(a) Calculus of language \( \mathcal{F} \).

\[
\begin{align*}
e &::= x \mid v \mid (e \; e) \\
\mathcal{E} &::= [\;]_{\mathcal{J}} \mid (\mathcal{E} \; e) \mid (v \; \mathcal{E}) \\
\tau &::= \text{JT}
\end{align*}
\]

\( \text{(ALL)} \quad \Gamma \vdash e : \text{JT} \quad \text{(FN}_J \text{)} \quad \frac{}{\mathcal{E}[(\lambda x . e) \; v] \rightarrow \mathcal{E}[e \; [v/x]]} \)

(a) Calculus of language \( \mathcal{J} \).

Figure 2.3: Toy calculi.
2.1.3 Lump embedding

The lump embedding is a form of basic language interaction in which values of one language are seen as opaque pointers in the other. This is similar to some FFI systems in which one of the languages has access to native values of the other language as pointers that can only be passed to the latter. An example of such a system is the type `ctypes.c_void_p`, which represents an opaque pointer in Python to a C value.

The syntax and semantics of each language are extended in Figure 2.4 to produce the lump embedding. $^\tau F J$ and $J F^\tau$ are syntactic language boundaries that indicate a change of language. The first one can be thought of as “$F$ expression inside, $J$ expression outside of type $\tau$”, and symmetrically for the other boundary. In these boundaries, $\tau$ is the type that $F$’s typing system considers its side of the expression to be.

A new type $L$ (for lump) is added to $F$, with values of this type being primitive (not re-imported from $F$) foreign values from $J$ imported via an $L F J$ boundary.

$$
\begin{align*}
\text{e ::= } & \cdots | \, ^\tau F J \ e \\
\text{v ::= } & \cdots | \, L F J \ v \\
\text{E ::= } & \cdots | \, ( ^\tau F J \ E ) \\
\tau ::= & \cdots | \, L \\
\end{align*}
$$

$$(\text{LUMP}) \quad \Gamma \vdash ^\tau F J \ e : \tau$$

$$(\text{F-LUMP}) \quad \mathcal{E}[ ^\tau F J ( J F^\tau v ) ] \to \mathcal{E}[ v ]$$

$$(\text{J-LUMP}) \quad \mathcal{E}[ J F^L ( L F J \ v ) ] \to \mathcal{E}[ v ]$$

$$(\text{F-ERROR}) \quad \mathcal{E}[ ^\tau F J ( v ) ] \to \mathcal{E}[ ( \text{error(“wrong-value”)}) ]), \quad \text{if } \tau \neq L \text{ or } v \neq J F^\tau v$$

Figure 2.4: Extensions to calculi $F$ and $J$ (Figure 2.3) to form the lump embedding.

$$
\begin{align*}
\text{bool F J ( ( \lambda x . x ) ( J F^{\text{bool}} \ true ) ) } & \to^{\text{bool}} F J ( J F^{\text{bool}} \ true ) \\
& \to \text{true}
\end{align*}
$$

Figure 2.5: Example of the lump embedding for $F$ (host) and $J$ (guest).
Figure 2.5 illustrates the lump embedding with a simple example, in which there is a language boundary \( \text{bool} \ F \ J \) containing a function application of the \( J \)-defined identity function to a boolean value inside another boundary.

### 2.1.3.1 Insufficiencies of the lump embedding

The main insufficiency of the lump embedding with respect to its implementation details was mentioned in §2.1.1: the impracticality of embedding a compiled language into another. The source of the compiled language needs to be analyzed and compiled, so using a string representation is problematic, as the compilation would be delayed until runtime.

Since JavaScript (for which \( J \) is a stand-in) is an interpreted language (so there is no static checking), it is embedded as a string in F\# source code. JavaScript programs are stored as source code and cannot be compiled, so no specific JavaScript features are lost due to this. However, this implies that JavaScript cannot contain F\# source code embedded as a string, which limits the creation of lumps to F#.

### 2.1.4 Natural embedding

The natural embedding provides a richer cross-language interoperability, by translating the values of one language into the other. This requires that each value in one language has a corresponding value in the other. The natural embedding further assumes an existing “translator” for primitive values, which are the rules \( J \rightarrow F \cdot \text{BOOL} \) and \( F \rightarrow J \cdot \text{FN} \) in Figure 2.6. These limitations are further discussed in §2.1.4.1.

The original languages \( F \) and \( J \) from Figure 2.3 are expanded in Figure 2.6 with additional syntax and reduction rules to result in the natural embedding. The boundaries used in the natural embedding differ from those in the lump embedding in that they translate values and perform dynamic first-order type checks. These type checks verify whether the value passing the boundary is of type \( \text{bool} \) or function \( (\tau_1 \rightarrow \tau_2 \text{ for any } \tau_1 \text{ and } \tau_2) \). If the check fails, an error is signaled (rules \( J \rightarrow F \cdot \text{B-ERROR} \) and \( J \rightarrow F \cdot \text{FN-ERROR} \) in Figure 2.6).

This preserves the type safety of \( F \):

**Theorem** (Type-safety for \( F \)). A well-typed expression \( e \) in \( F \), \( e \vdash \tau \), doesn’t get “stuck”: either \( E[e] \rightarrow E[v] \), \( E[e] \rightarrow E[\text{error}] \) \( J \) is the only source of errors), or \( e \) diverges.

**Proof.** Proof by a standard argument, similar to [1] §3.2. \( \square \)
2.1. EMBEDDINGS

\[
e ::= \cdots \mid F J G^\tau e
e ::= \cdots \mid G J F^\tau e
\]

\[
\mathcal{E} ::= \cdots \mid F J G^\tau \mathcal{E}
\mathcal{E} ::= \cdots \mid G J F^\tau \mathcal{E}
\]

\[
(F-\text{TRANS}) \quad \frac{\Gamma \vdash F J G^\tau e : \tau}{\Gamma \vdash F J G^\tau e : \tau}
\]

\[
(J-\text{TRANS}) \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash G J F^\tau e : J T}
\]

\[
(J-\text{to-}F-\text{Bool}) \quad \frac{}{\mathcal{E}[F J G^{\text{bool}}(b)] \to \mathcal{E}[b]}
\]

\[
(F-\text{to-}J-\text{Bool}) \quad \frac{}{\mathcal{E}[G J F^{\text{bool}}(b)] \to \mathcal{E}[b]}
\]

where \((b, b) \in \{ (\text{true, true}), (\text{false, false}) \}\)

\[
(J-\text{to-}F-FN) \quad \frac{}{\mathcal{E}[F J G^{\tau_1 \to \tau_2}(\lambda x. e)] \to \mathcal{E}[\lambda x : \tau_1. F J G^{\tau_2}((\lambda x. e)(G J F^{\tau_1} x))]}
\]

\[
(F-\text{to-}J-FN) \quad \frac{}{\mathcal{E}[G J F^{\tau_1 \to \tau_2}(\lambda x : \tau_1. e)] \to \mathcal{E}[\lambda x. G J F^{\tau_2}((\lambda x : \tau_1. e)(F J G^{\tau_1} x))]}
\]

\[
(J-\text{to-}F-B-\text{error}) \quad \frac{}{\mathcal{E}[F J G^{\text{bool}}(v)] \to \mathcal{E}[F J G^{\text{bool}}(\text{error("not-bool")})],}
\]

if \(v \notin \{\text{true, false}\}\)

\[
(J-\text{to-}F-FN-\text{error}) \quad \frac{}{\mathcal{E}[F J G^{\tau_1 \to \tau_2}(v)] \to \mathcal{E}[F J G^{\tau_1 \to \tau_2} (\text{error("not-fn")})],}
\]

if \(v \neq \lambda x. e\), for any \(x\) or \(e\).

Figure 2.6: Extensions to languages \(F\) and \(J\) (Figure 2.3) to form the natural embedding.

As shown in Figure 2.6, the conversion of booleans is trivial, only requiring us to modify the byte representation of the value to be translated (here indicated by changing the color of the syntactic boolean term). Function values are more interesting: when translating a function \(f\), we create a function \(g\) with argument \(x\) in the native “target” language. \(g\) will translate \(x\), apply \(f\) and then translate back the resulting value.

We signal errors when the untyped language \(J\) provides a value that does not adhere to the type specification in the guarded boundary. For instance, this occurs when the \(F\) side of the boundary expects a function type but receives a boolean value (reduction \(J-\text{to-}F-FN-\text{error}\)).
CHAPTER 2. PREPARATION

2.1.4.1 Insufficiencies of the natural embedding

The fact that JavaScript is embedded in F# as strings has no effect in the natural embedding implementation, since the values are translated, rather than wrapped inside lumps.

A limitation of the natural embedding as presented here is the fact that real-life languages don’t have values that exactly match (cf. §1.3.2). This leads to more complex conversion strategies, where one single value can perform different reductions depending on the expected type on the other side of the translation boundary (type-mapped). For instance, this occurs when translating JavaScript Numbers (which are floating-point numbers) to F#, which supports both ints and floats.

The existing translator for primitive values mentioned above is provided by Platform Invoke (P/Invoke) and embedding and projection algorithms (described next).

2.1.5 Type-indexed embedding and projection algorithms

Type-indexed embedding/projection pairs are used to translate values from two languages, given that the host language has access to an interpreter of the guest. This assumes the role of the “translator” for primitive values mentioned in §2.1.4.

Lua-ML is a system in production for the C– compiler [10, 11] that is based on this approach, with OCaml being the host language and Lua being the guest. For each type \( \tau \), a \( \tau\.embed \) and \( \tau\.project \) pair is defined, which translates a value of type \( \tau \) from OCaml to Lua (embed) and from Lua to an OCaml \( \tau \) value (project). These pairs are then used by instantiating them for the appropriate type and passing them a value to translate.

Listing 2.1 shows the implementation of the embedding/projection pair for the bool type (note that Lua has no specific boolean values, it instead considers Nil to be false and any other value to be true): bool.embed takes an OCaml bool value and, if it is true, it is embedded as the string "t" in Lua; otherwise it turns into Nil. bool.project takes a Lua value and checks whether it is Nil (which maps to false), or any other value (which converts to true). An example use of the pair bool is shown in lines 3–5 of Listing 2.1.

```ocaml
let bool = { embed = (fun b -> if b then String "t" else Nil); 
  project = (function Nil -> false | _ -> true) }

let t = true
let t_from_lua = bool.project (bool.embed t)
assert (t = t_from_lua)
```

Listing 2.1: bool pair from Lua-ML (lines 407–408 from the source file luavalue.nw [12]).
2.2 V8 JavaScript engine

In this section, we present the inner workings of the V8 JavaScript engine [13], as some familiarity is required in order to follow §3.

The purpose of a JavaScript engine is to read and execute JavaScript source code. V8 is a high-performance JavaScript engine that was first released in September 2008 by Google. It provides a C++ API to compile and execute scripts, handle errors, etc. The API provides a set of classes that correspond to JavaScript types such as `Number` or `Object`. Instances of these classes can be wrapped to produce handles. Handles are references to JavaScript objects’ locations on the heap, and there are two types:

- **Local handles** are held on the V8 stack (whose scope is defined by the current C++ stack) and are deleted when the destructor is automatically invoked. When this occurs, V8’s garbage collector is free to deallocate objects previously referenced by the handles in the handle scope. If JavaScript is the guest language, V8 procedures must return the control flow to the host language (F#) at some point. Consequently, this type of handle is not particularly useful for cross-language communication.

- **Persistent handles** are held on the heap and the user must specifically dispose of them. Persistent handles can be weakened, which signals to the garbage collector that if no other persistent handles refer to the value in question, it may be collected. We discuss in §3.3.8 the memory management of these handles from F#.

V8 exposes contexts: execution environments that allow separate, independent JavaScript applications to run in a single instance of V8. This allows a user to modify built-in JavaScript functionality by changing the global object, which contains built-in functionality available to a JavaScript program, such as `Math`, `String`, `Infinity`, etc. [14, §15].

![Figure 2.7: V8 handles overview, from the V8 documentation][13]
2.3 Software engineering techniques

This section outlines the software engineering techniques followed for the design, implementation and testing of this project.

Figure 2.8: Gantt chart illustrating the project schedule and its completion as of January 18, 2013.
2.3.1 Development methodology

An iterative approach was taken when working on this project. The initial planning (see a partially completed Gannt chart in Figure 2.8) and requirements analysis were performed as described in §2.3.2. The design of specific components, its implementation (§3) and testing (§4.2) were carried out in several bi-weekly iterations (described in Appendix E), which produced prototypes of the system for each iteration. This approach was used to reduce risk due to the unfamiliarity at the beginning of the project with the theory and tools/languages used.

A high degree of modularization was planned in order to provide enough flexibility and loose coupling, required especially for the iterative approach. This was achieved by making use of the F# module system; for instance, all functionality provided by the V8 JavaScript engine was abstracted in the JSEngine module, so that a change in the JavaScript engine to be used (alternatives such as Mozilla Rhino [15] were considered) could be made without altering any other module.

2.3.2 Requirements analysis

There are two deliverables in this project: an implementation of the lump embedding and the natural embedding. The former is mainly done for completeness with respect to the Ph.D. dissertation this project is based on [1], while the latter emerges as a more powerful and novel technique. Both types of embedding may be practical for the user story described next.

As we can see from Figure 2.9, the use case of this system is a software engineer who wants to use both F# and JavaScript as two of his programming languages in a project. This may be because of the need to interact with certain existing libraries unavailable in a language (e.g., use D3.js in F# to manipulate graphical interfaces based on data), the desire to have a JavaScript engine embedded into the application (e.g., a game engine that allows designers and end-users to customize the behavior of the system without recompiling the whole project [16]) or simply because some tasks are better performed in another language (e.g., parsing command line arguments in a strictly typed language like F# is not as convenient as performing this task in JavaScript).
Derived from the use case, the requirements (expanding those listed in the original proposal, see Appendix E) of this project are:

- The lump embedding implementation must preserve the state of both runtime environments (F# and JavaScript) to allow for a reliable cross-language communication.
- The natural embedding implementation must provide an exact translation of values. When this is not possible, an approximation should be produced.
- The resulting system should not produce a significant overhead executing time over the monolingual runtimes.
- A convenient syntax for multilanguage programming must be implemented.
- F# is a richer language than JavaScript, so it will be a many-to-one mapping of F# types to JavaScript types. We require that most values in F# can be translated to JavaScript.
2.3.3 Practical preparation

The candidate has encountered the ML family of languages earlier in his studies, but he needed to become familiar with F#-specific constructs (e.g., active patterns, quotations). The candidate was not familiar with JavaScript at the start of this project and became acquainted with its features (e.g., prototype-based inheritance). The candidate also held basic knowledge of C++, but no previous experience using V8 or MonoDevelop (IDE).

The documentation and examples available for V8 (§2.2) are notoriously obscure, and it took a considerable amount of time to grapple with its operation.

2.3.4 Development tools

2.3.4.1 Development environment

Even though F# is a language developed by Microsoft, the Mono project [17] —which provides a cross-platform .NET framework— is considered to be mature enough for production code. Hence, it was used to compile and run F# code, which was written in Emacs (using fsharp-mode) and MonoDevelop. Dependencies between files were handled using the Unix tool Make.

The choice of programming languages for this project is justified as follows:

- F# is an ML-like language that has suitable similarities to language $\mathcal{F}$ (Figure 2.1a), used to describe the embeddings: first-class functions, strongly and statically typed. It is further open-source and cross-platform, and it is increasingly used in finance [18], gaming, web programming [19], etc., partly due to its easy interoperation with other .NET languages. This allows the resulting system of this project to be used in a wide range of situations.

- JavaScript is a very popular language used mainly in the client-side (web browser), but also in server-side applications. Its bad reputation is mainly due to differing DOM APIs in web browsers, misuse of the language, and some design errors (e.g., programming model based on global variables). However, JavaScript has many good parts [20]: first-class functions, powerful object literal notation, ubiquitous run-times, etc.

Furthermore, many authors consider JavaScript as Lisp or Scheme in C-like syntax, which makes JavaScript a good choice for this project —Matthews’ dissertation describes the operational semantics of the lump and natural embeddings for an ML-like and an Scheme-like calculi. This was a reassurance that an implementation for
F# and JavaScript was feasible, although some modifications and extensions to the theory were required.

- C++ procedures were implemented to provide the bridge between F# and JavaScript. Its use is justified by the fact that the V8 JavaScript engine is implemented in C++ and the .NET framework provides P/Invoke, a system for interacting .NET (F#) with native code.

2.3.4.2 Version control and backup

Git was used as the version control system for both the source code and the dissertation files, each in a separate repository. The Git repositories were hosted on a private repository on Bitbucket\(^1\) and replicated to the Desktop Services provided by the Computer Laboratory\(^2\).

These remote repositories served as a hosting solution (so the candidate was able to work on this project while not having access to his machine), and as a backup system. The Git repository for the source code was also useful when writing the dissertation, as it served as a work log.

2.4 Summary

The planning and research work undertaken before implementing the project has been described. The schedule has not been modified from that in the project proposal, which will be followed using an iterative approach. The research work includes the lump and natural embedding, and the type-indexed embed/project algorithms used to implement the natural embedding. Therefore, the concrete deliverables are two components: the lump and natural embeddings.

\(^1\) with hostname www.bitbucket.org
\(^2\) with hostname linux.cl.ds.cam.ac.uk
Chapter 3

Implementation

This chapter describes how the theory and algorithms explored in §2 were implemented for MiXture, the system the candidate implemented for the programming languages F# and JavaScript. The implementation of the two main components (the lump and natural embeddings) of this project are discussed in detail.

3.1 High-level code structure

![Diagram of MiXture's high-level design](image)

We begin by introducing the overall architecture used in the implementation, illustrated in Figure 3.1. JSEngine contains the functions imported into F# from V8Utils. These functions provide the basis for the interaction with JavaScript, which are then used in JSUtils to provide more suitable functions to use when implementing the lump embedding (LEmbedding; §3.2) and the natural embedding (NEmbedding; §3.3).
3.2 Lump embedding

As we saw in §2.1.3, the lump embedding introduces the concept of lumps, which are opaque pointers to values from a different language than the current environment. Since JavaScript cannot inspect the F# lumps and vice versa, there is no restriction to the type of values that can be passed between languages.

In the following sections, we summarize the implementation approach followed.

3.2.1 Implementation of lumps

In MiXture, there are two classes that represent lumps:

- 'a FSLump: an F# value of type 'a that has been embedded into JavaScript,
- JSLump: a JavaScript handle value in F#.

Due to the insufficiencies identified in §2.1.3.1, an 'a FSLump value denotes an F# value wrapped in a $\alpha FJ(JF^\alpha)$ boundary, rather than a simple $JF^\alpha$ (F# inside, JavaScript outside). This is because it symbolizes an embedded F# value in JavaScript in an F# environment, and hence the double wrapping. 'a FSLumps can be passed to the JavaScript environment, producing a $JF^\alpha(\alpha FJ(JF^\alpha))$, which is semantically equivalent to the expected $JF^\alpha$.

The JSLump class closely corresponds to a value inside a $L FJ$ boundary (JavaScript inside, F# outside, seen with type L —lump).

Objects of the class JSLump reference a JavaScript value $v$ via a nativeint value, which points to the space allocated for $v$ in the heap of V8. FSLumps also contain a Pointer attribute in order to allow JavaScript programs to reference this type of lump. Memory management of these lump values is discussed in §3.2.3.

3.2.2 Cross-language communication

Cross-language communication is the main purpose of a multilanguage system. This interaction is formally defined for the lump embedding in §2.1.3 via its operational semantics. Here, the top-level functions are described.
3.2. LUMP EMBEDDING

3.2.2.1 Function application of FSLump instances in JavaScript

This functionality allows a user to make a function call with the value wrapped inside an FSLump from JavaScript. A use example is given in Listing 3.1 where a CPU intensive operation is offloaded to F#.

Listing 3.1: Computer vision example of foreignApply, where an F# function is wrapped in a (int list -> int list) FSLump and is applied to an image I and a convolution kernel K. The types of the variables are given in an ML-like specification in a comment, as recommended by Felleisen [21].

A function evaluator is defined with the purpose of evaluating the application of an F# function inside an (‘a -> ’b) FSLump to other Lump values (both JSLump and FSLump). The pseudocode can be found in algorithm 3.1.

- a delegate is created for evaluator and this is registered in V8. This action is performed when the lump embedding library is loaded in an F# application.
- Lumps are transmitted to V8 for a JavaScript application to control.

The pseudocode can be found in algorithm 3.1.
CHAPTER 3. IMPLEMENTATION

Figure 3.2: Interoperation between F#, V8 and JavaScript for the lump embedding.

Algorithm 3.1 Evaluator function for a function wrapped in a FSLump.

Input:
- \( f \): an \('a \rightarrow \ 'b\) FSLump for some \('a\) and \('b\); is the function to be applied.
- \( \text{args} \): a Lump list, which represents the list of arguments.
- Note that the combination of \( f = \ 'a\) FSLump (not a function type) and \( \text{args} = []\) is valid. See comment in line 18.

Output: The function \( f \).Value applied to the arguments in \( \text{args} \).

```
function EVALUATOR(f, args)
    function EVALUATE_LUMP(f, arg)
        domain ← typeof(\'a)  \( \triangleright \) \('a\) is the domain of \( f \)
        range ← typeof(\'b)  \( \triangleright \) \('b\) is the range of \( f \)
        if arg is a JSLump then
            actual_in_type ← JSLump
        else if arg is a \('c\) FSLump then
            actual_in_type ← \('c\)
        if domain \( \neq \) actual_in_type then ERROR("Type mismatch")
        if arg is a JSLump then
            val ← OBJECT(arg)
        else if arg is a \('c\) FSLump then
            val ← arg.Value
        return f(val)
    if args is a list \( x :: xs \) then
        return EVALUATOR(EVALUATE_LUMP(f, x), xs)
    else if args is the empty list [] then
        return f \( \triangleright \) \( f \) acts as an accumulator for curried function application
```
3.2. LUMP EMBEDDING

3.2.2.2 Function application of JSLump instances in F#

This section explains the process of invoking —from F#— a JSLump value that points to a JavaScript function. This functionality is provided by the function `applyJSLump: JSLump -> Lump list -> Lump list`, whose first argument is the JavaScript function, the second is the argument list (either ’a FSLump or JSLump) and returns a result list.

This function passes the Pointer attribute of the first argument to a C++ implemented procedure, which uses the V8 API to invoke the JavaScript function.

3.2.3 Memory management

§3.3.8 explains memory management in MiXture in more detail. It uses the garbage collectors for F# and V8 to save the programmer from doing manual memory management. Here we give a brief overview for LEmbedding.

The class `JSValue` holds a reference to an unmanaged resource (not handled by the F# runtime). We obtain automatic memory management by making `JSValue` implement the `IDisposable` interface and a `Finalize` method. The finalizer then registers a message to V8 when the JSLump value is about to be garbage collected in F#.

We mentioned in §3.2.1 that ’a FSLump contains a Pointer attribute to allow JavaScript programs to reference them. When an ’a FSLump is passed to V8, it is inserted into `FSValuesStorage`, a class whose underlying implementation is a dictionary. `FSValuesStorage` maps IdTypes (the Pointer attribute) to FSLumps. Figure 3.3 illustrates the class hierarchy and overview of the lump embedding.

![Figure 3.3: UML class diagram for lumps.](image-url)
3.3 Natural embedding

This part of the project is of more practical use than the lump embedding, and hence more time was allocated to its implementation.

As described in §2.1.4, the natural embedding allows the user to translate values between two programming environments. The ability to translate non-primitive values leads to the possibility of an unlimited number of types to be converted between languages, as it will be shown in later subsections.

The natural embedding has been extended to deal with new types of values. The new syntax, typing rules, reductions and type-safety proofs are given in Appendix A.

3.3.1 Representation of a JavaScript handle in F#

The module NEEmbedding defines the important type JSValue, which holds a pointer to a JavaScript value on the heap of an instance of V8. This type has other members that deal with function application and object properties access (§3.3.10), and memory management members (§3.3.8).

3.3.2 Embedding/projection pairs

In this section, we summarize the use of embedding/projection pairs and describe the differences between the approach taken in this project with respect to that of existing work. As far as the author is aware, this is an original approach.

An embedding/projection pair is represented as an (’a, ’b) ep record type, similar to what is found in [5]. The definition of (’a, ’b) ep is shown in Listing 3.2. ’a is the type of the value to be embedded into JavaScript, and ’b is the type in F# for native JavaScript values. Note that in this project, all pairs of this kind have instantiated ’b to JSValue. That is, an F# value of type ’a is embedded to result in a value of type JSValue. A JavaScript value in F# (of type JSValue) is projected to be an F# value of type ’a. There is one (’a, JSValue) ep pair for each type to be converted.

```
type (’a, ’b) ep = { embed : ’a -> ’b; project: ’b -> ’a }
```

Listing 3.2: Definition of the record type (’a, ’b) ep, an embedding/projection pair used to translate values between F# and JavaScript.
The originality of this implementation resides in the abstraction of which type is being embedded/projected. In order to embed/project a value, it is necessary to know its type. Consequently, Lua-ML \[5\] and Benton \[3\] suggest a verbose syntax that selects the embed/project pair by its name (which coincides with the type being converted):

- For primitive values: \texttt{type.\{embed | project\} v}, such as \texttt{bool.project b}.
- For non-primitive values: \texttt{create_pair(type).\{embed | project\} v}, such as \texttt{(func (int **-> result int)).embed (fun x -> x+1)}.

While this syntax is dense compared to other systems (cf. P/Invoke, ctypes) and allows for reduced glue code, MiXture uses meta-programming in order to reduce the type annotations the user needs to provide. Embedding/projection pairs are also defined in this project, but they are not visible to the user. Instead, two top-level functions are defined to perform the heavy-lifting of dispatching each value according to its type:

- \texttt{embed:obj->JSValue}. This function takes an argument of type \texttt{obj} (all F\# types are subtypes of \texttt{obj}) and produces a \texttt{JSValue}. The domain is not a polymorphic type \texttt{\'}a\texttt{\'} because it inspects the type of the argument \texttt{x} by calling \texttt{x.GetType()}, and according to the result, dispatches \texttt{x} to the corresponding embedding/projection pair. This allows the user of MiXture to simply use the syntax \texttt{embed v}.

\texttt{embed} always succeeds for the supported types, since JavaScript is loosely typed and hence there are no type expectations in its environment.

- \texttt{project<\'}T\texttt{\'>:JSValue->\'}T}. This function takes as an argument a native JavaScript value, denoted by the type \texttt{JSValue} in F\#, and returns the corresponding F\# value of type \texttt{\'}T\texttt{\'}.

In the cases in which the type system cannot infer \texttt{\'}T (e.g., in a \texttt{let}-binding in an interactive session), there are two possibilities:

- the programmer includes an ordinary type annotation (unlike in Lua-ML and Benton’s systems, which require non-native type annotations). An example is:

\begin{verbatim}
let name:string = project query_result.
\end{verbatim}

---

\[1\] \texttt{List.reduce<\'}T\texttt{\'>\:\{\'}T->\'}T\texttt{\'>\->\'}T \texttt{list} \texttt{-> \'}T \texttt{is the famous traversing function foldl without an initial value.}
MiXture inspects the type of the JavaScript value via pattern matching (with active patterns) and projects to the best guessed type in F#. The previous example without a type annotation (let name = project query_result) will still assign the type string to the value name, provided query_result points to a JavaScript string. This approach only works for primitive types, as JavaScript cannot provide enough information for other types such as functions and objects.

project can fail (and raise a ProjectionException) if the JavaScript value cannot be projected to the expected F# type.

Active patterns\(^2\) were used to provide an easy way to access the F# representation of a JavaScript value. This allows the user to perform pattern matching with a JSValue, which was heavily used in the implementation of the embedding/projection pairs and the top-level function project. Some of the active patterns provided by MiXture can be seen in use in Listing 3.3, where the argument jh: JSValue points to a JavaScript handle.

Listing 3.3: Illustrating the use of active patterns to pattern match a JavaScript handle. Note that not all active patterns have been included.

3.3.3 Notation

In the following sections, we give some translation rules for embedding and projecting values. We define a binary function

\[
\text{embed} : \text{F# values} \times \text{JavaScript values} \rightarrow \subseteq \text{F# values} \times \text{JavaScript values}
\]

so that \(v_{\text{F#}} \xrightarrow{\text{embed}} v_{\text{JS}}\) is read “the F# value \(v_{\text{F#}}\) is embedded to result in the JavaScript value \(v_{\text{JS}}\).”

\(^2\)Active patterns are an F# construct that allows the programmer to perform pattern matching against values that couldn’t otherwise be expressed in a pattern match rule.
We define a family of similar relations for projecting values from JavaScript to F#. These relations are indexed by the type $\tau$, as seen in the F# boundary, in order to make them functional (due to type-mapping, cf. §1.3.2):

$$
\text{project}_\tau : \text{JavaScript values} \times \text{F# values of type } \tau.
$$

(3.2)

These relations are a clearer representation of reduction rules in terms of F# and JavaScript, as opposed to $\mathcal{F}$ and $\mathcal{J}$ (§2.1):

$$
\text{embed} = E[GJF^\tau] \rightarrow \text{(3.3)}
$$

$$
\text{project}_\tau = E[FJG^\tau] \rightarrow \text{(3.4)}
$$

All translation rules are included in Appendix B and mapping of the types supported by MiXturer can be seen Table B.1.

### 3.3.4 Primitives

Primitive types are usually standard across most programming languages: in F#, these include int, float, string, bool, unit, etc. [23], and they match JavaScript primitive values: Numbers, strings, booleans, undefined, and null [24]. The majority of the byte translation was performed by P/Invoke, which allows .NET managed code to call unmanaged procedures that are implemented in a DLL. These procedures need to be unmanaged C++ due to V8, and can be divided into two categories:

1. Wrappers for the V8 API. These include extracting primitive values from JavaScript handles, such as extractFloat, and some auxiliary procedures such as setElementArray. These are required for two reasons: i) the V8 API is not prepared to be called using P/Invoke; and ii) to allow independence of the JavaScript engine used in the project, making the substitution by another engine a matter of writing a matching wrapper interface.

2. Full-blown procedures. These consist of more complex procedures such as executeString or applyFunctionArr, which must take care of exceptions (§3.3.9).

The translation for bool values is simple and similar to that shown when describing Lua-ML in Listing 2.1 except that more cases need to be considered, as any JavaScript value can be used as a boolean value (boolean, falsy and truthy values [24 §3.3]).

The case of ints and floats is more interesting, since there is no exact correspondence with JavaScript Numbers. JavaScript only provides floating point arithmetic operations
(double-precision 64-bit format IEEE 754 values). There are three possible cases when the value to be projected is a JavaScript Number and the expected type is either int or float, as shown in the translation rules of Figure 3.4. The first rule corresponds to an integral number in JavaScript being successfully projected into an F# int. The second rule exhibits erroneous behavior due to a type mismatch — \( f_{JS} \) is not a whole number. The third rule describes the translation of a floating-point number in JavaScript to a float value in F#.

Note that all other cases where the type to be projected is not int or float will raise an exception as in rule Number-error.

\[
\frac{f_{JS} \text{ project} \rightarrow n_{F#}}{\text{if } \text{truncate}(f_{JS}) = f_{JS} \text{ and } n_{F#} = \text{truncate}(f_{JS})}
\]

\[
\frac{f_{JS} \text{ project} \rightarrow \text{Error}}{\text{if } \text{truncate}(f_{JS}) \neq f_{JS}}
\]

\[
\frac{f_{JS} \text{ project} \rightarrow f_{F#}}{\text{Number-to-float}}
\]

Figure 3.4: Translation rules for JavaScript Number. The subscript in the numeric values indicates the programming language they belong to, where JS stands for JavaScript. \text{truncate} sets to zero the decimal digits of a floating point number: e.g., \( \text{truncate}(3.14) = 3.0 \); \text{round} maps a non-negative floating point number \( f \) to \text{floor}(f): e.g., \( \text{round}(3.14) = 3 \); and a non-positive number \( f \) to \text{ceiling}(f): e.g., \( \text{round}(-3.14) = -3 \), where \text{floor} and \text{ceiling} map floating point numbers to the smallest following integer and the largest previous integer, respectively. Error stands for raising a ProjectionException.

### 3.3.5 Function values

The ability to translate function values is crucial for a transparent cross-language communication system. This allows the treatment of foreign functions as if native in both languages, being able to make use of functional programming features such as first-order foreign functions.

Even though both F# and JavaScript have first-class functions, they behave differently. From a semantic point of view, all F# functions are curried (polyadic functions are a
3.3. NATURAL EMBEDDING

nested series of unary functions; non-curried functions can be simulated by the use of tuple values), whereas functions in JavaScript are traditionally written in non-curried form (although JavaScript also supports currying). As a result, in F#, a partially applied function creates a closure with its free variables bound to the supplied arguments; JavaScript, however, adjusts a partially applied function, meaning that missing arguments take the value undefined. This significant difference in the semantics is dealt with when embedding/projecting function values, as described below.

The implementation of function translation follows the reductions \( J \rightarrow F \rightarrow Fn \) (Proj-Func) and \( F \rightarrow J \rightarrow Fn \) (Embed-Func) from Figure 2.6 (Figure 3.5).

3.3.5.1 Embedding functions

The embedding rule for function values is given in Figure 3.5: an F# function \( f \) is embedded by creating a JavaScript function \( g \) that projects its arguments, applies \( f \) to them, applies \( \text{embed} \) to the result from \( f \) and returns it. V8 allows the creation of JavaScript functions at runtime via function templates, which represent the blueprint of a single function. Function templates can be constructed in C++ by providing a pointer to a function whose argument is a reference to a constant v8::Arguments object, and returns a v8::Handle. This value is originally created in F#, which is a function \( g: \text{JSValue} \rightarrow \text{JSValue} \) that processes the arguments by calling wrapper procedures for V8 and returning the result of \( \text{embed} (f \ (\text{project processed_args})) \).

The design decision of not uncurrying F# functions when embedding them was taken for two reasons:

1. application of curried JavaScript functions is not as syntactically awkward as defining them: calling a curried function requires surrounding each argument with parentheses, as in \( \text{curried_js_function(arg1)}(\text{arg2}) \); whereas defining one must be of the form
   \[
   (\text{function(arg1)} \ {\text{return (function(arg2)} \ {\text{...}} \ )})).
   \]

2. to allow the more powerful curried form in JavaScript, supporting partial application.

3.3.5.2 Projecting functions

Projecting a JavaScript function \( h \) involves the reverse of the process described in the previous paragraph, and is formally specified in Figure 3.5. However, in order to support currying (and hence obtain partial application) of JavaScript polyadic functions when projected into F#, MiXture accumulates the arguments for \( h \) until its arity matches the
number of arguments collected. This is performed by using the F# type associated with the projected value (inferred or annotated) and illustrated in algorithm 3.2: check whether the expected result value is a function or non-function value.

This means that MiXture will project a function according to the type specified by the user/type system, whether the projected function will be curried (separating the argument types with the function constructor “->”) or not (separating the argument types with the tuple constructor “*”).

In Listing 3.4 we can see that a ternary JavaScript function is projected as a curried function in F#, which allows partial application (as in line 9).

```fsharp
let surround_str: string->string->string =
    "(function(beginning, end, str) {
        return beginning + str + end;
    })"
 |> JSUtils.execute_string
 |> project
// surround_str is now an F# curried function!
let angle_surround = surround_str "<" ">
printfn "%s" <| angle_surround "Hello, world!"
// ==> "<Hello, world!>
```

Listing 3.4: Projecting a ternary JavaScript function into F#, creating a curried function.

The use of reflection avoids the need to use custom functions to annotate the projected type (such as the Lua-ML functions **-> to denote function type, func to construct an embedding/projection pair or result to indicate the type of the range of a function).
Algorithm 3.2 Projection algorithm for functions.

Input:
- $type$: a System.Type reflected type that indicates the function type to be obtained. This is supplied by project.
- $f$: a JSValue that contains a pointer to the JavaScript function to be projected.

Output: $result$ is an F# function equivalent to $f$.

```fsharp
1 function PROJECT_FUNC(type, f)
2     range ← RANGE(type)
3     if range is a function type then
4         function RESULT(arg)
5             PROJECT_range(fun t => F(EMBED(arg :: t)))
6     else
7         function RESULT(arg)
8             PROJECT_range(F(EMBED([arg])))
9     return result
```

3.3.6 Collections

This section provides a sample proof and describes the implementation for embedding and projecting arrays, lists and tuples. Lists and tuples are important data structures in functional programming languages, and are widely used in F#. Hence, even though the translation of collections was an extension to the core of the project, it was deemed of high priority and completed as soon as the core was in a stable state.

Arrays are also a very important data structure, as they are the underlying implementation of most other data structures. JavaScript provides arrays only, whereas F# provides arrays, lists and tuples. This is an instance of why the natural embedding for JavaScript and F# is type-mapped—a JavaScript array can be projected both as an array or a list (if all elements are of the same type), or a tuple (if not all elements are of the same type) in F#.

3.3.6.1 Sample proof of type-safety

In this section, we give a sample proof of type-safety for MiXture. Due to space constraints, the collected definition (syntax, typing rules, semantics) of our model calculi $\mathcal{F}$ and $\mathcal{J}$ and the full type-safety proofs for the new cases introduced in this dissertation can be found in Appendix A.
We begin by extending the reduction rules of the natural embedding from Figure 2.6 to introduce lists to the toy calculi from §2.1.

\[
\begin{align*}
  e ::= & \cdots | e \cdot e | \text{head } e | \text{tail } e & e ::= & \cdots | e \cdot e | \text{head } e | \text{tail } e \\
  v ::= & \cdots | v \cdot v | \text{nil} & v ::= & \cdots | v \cdot v | \text{nil} \\
  \mathcal{E} ::= & \cdots | \text{head } \mathcal{E} | \text{tail } \mathcal{E} | v \cdot \mathcal{E} | \mathcal{E} \cdot e & \mathcal{E} ::= & \cdots | \text{head } \mathcal{E} | \text{tail } \mathcal{E} | v \cdot \mathcal{E} | \mathcal{E} \cdot e
\end{align*}
\]

\[
\begin{align*}
  \text{(CONS)} & \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \cdot \text{list} \quad \rightarrow \quad \Gamma \vdash e_1 \cdot e_2 : \tau \cdot \text{list} \\
  \text{(NIL)} & \quad \Gamma \vdash \text{nil} : \tau \cdot \text{list} \\
  \text{(HEAD)} & \quad \Gamma \vdash e : \tau \cdot \text{list} \quad \rightarrow \quad \Gamma \vdash \text{head } e : \tau \\
  \text{(TAIL)} & \quad \Gamma \vdash e : \tau \cdot \text{list} \quad \rightarrow \quad \Gamma \vdash \text{tail } e : \tau \cdot \text{list}
\end{align*}
\]

\[
\begin{align*}
  \text{(J-TO-F-LIST)} & \quad \mathcal{E}[F J G^{\tau} \cdot \text{list}(v_1 :: v_2)] \rightarrow \mathcal{E}[(F J G^{\tau}(v_1)) :: (F J G^{\tau} \cdot \text{list}(v_2))] \\
  \text{(F-TO-J-LIST)} & \quad \mathcal{E}[G J F^{\tau} \cdot \text{list}(v_1 :: v_2)] \rightarrow \mathcal{E}[(F J G^{\tau}(v_1)) :: (F J G^{\tau} \cdot \text{list}(v_2))]
\end{align*}
\]

Figure 3.6: Extensions to Figure 2.6 to include lists.

We now provide the case for type preservation for $\mathcal{F}$ lists.

**Theorem** (Type preservation). If $\Gamma \vdash e : \tau$ and $\mathcal{E}[e] \rightarrow \mathcal{E}[e']$ then $\Gamma \vdash e' : \tau$.

**Proof.** We prove type preservation by rule induction on reduction derivations.

**Case (J-TO-F-LIST).** Assume

\[
\Gamma \vdash F J G^{\tau} \cdot \text{list}(v_1 :: v_2) : \tau
\]

The last rule in the typing derivation must have been (F-TRANS), and hence $\tau = \tau' \cdot \text{list}$.

Considering the reduction of $F J G^{\tau'} \cdot \text{list}(v_1 :: v_2)$ according to (J-TO-F-LIST), by (F-TRANS) we have

\[
\begin{align*}
  \Gamma & \vdash (F J G^{\tau'}(v_1)) : \tau' \\
  \Gamma & \vdash (F J G^{\tau'} \cdot \text{list}(v_2)) : \tau' \cdot \text{list}
\end{align*}
\]

We can then use (3.5) and (3.6) with (CONS) to derive

\[
\Gamma \vdash (F J G^{\tau'}(v_1)) :: (F J G^{\tau'} \cdot \text{list}(v_2)) : \tau' \cdot \text{list}
\]
### 3.3. NATURAL EMBEDDING

#### 3.3.6.2 Implementation

The implementation of embedding F# arrays and lists is eased by both data structures implementing the interface IEnumerable. We create a JavaScript array of the same length as the F# value and then recursively embed all the elements.

Projecting JavaScript arrays is more troublesome: we need to dynamically produce a value whose type is not a primitive. MiXture defines two functions, called by `project<T>` when the expected type is `τ[]` or `τ list`, for some type `τ`. These two functions take as the first argument the reflected type for `τ` — the type of each of the elements in the collection — and, using reflection, create the appropriate array and list types.

The projection rule for F# lists is given in Figure 3.7: we require every element of the JavaScript array to be able to be projected with the same type `τ`.

\[
\begin{align*}
(v_1, v_2, \ldots, v_k) \xrightarrow{\text{project}} [v_1; v_2; \ldots; v_k]
\end{align*}
\]

Figure 3.7: Project rule for F# lists.

#### 3.3.7 Records

F# record values resemble JavaScript objects, since both are key-value collections. This is the only case in which the JavaScript side has more features than its F# counterpart, since JavaScript objects support prototypal inheritance, unlike F# records, which do not support inheritance. Yet another difference is that the JavaScript version used in this project (ECMAScript 5) also supports property attributes (writable — not read-only —, enumerable — can be enumerated in a for ... in loop — and configurable — can be deleted —), while entries in F# records only have one attribute: mutable (same as writable).

It is straightforward to translate an F# record to a JavaScript object. We recursively embed the entries in the record and create a JavaScript object with properties of the same name as the record labels, and set them to the embedded value corresponding to each entry. The JavaScript objects properties are set to enumerable and configurable (cannot be expressed in a record), and writable only if the record field was defined mutable. This is shown in Figure 3.8.
Projecting JavaScript objects has more complications:

- F# does not support anonymous record types (also known as record literals), so a matching type definition must be provided prior to projecting an object. The type definition must match in the names of the fields of the object being projected.

- It is a lossy translation. Some of the information is lost when performing the projection: a) the prototype chain for inheritance, and b) the property attributes enumerable and configurable cannot be represented in F# records.

### 3.3.8 Memory management

Memory management is an important aspect of multilanguage systems. Most multilanguage systems require the memory to be explicitly managed, in order to avoid garbage collectors removing objects that the other environment is not aware of. This is especially true for FFIs, such as the JNI, in which one of the languages is manually managed.

Both F# and JavaScript are automatically managed languages, and MiXture uses this fact to avoid the need to manually manage memory (de-)allocation. The host environment is the F# runtime, so F# values for which no memory is shared with JavaScript do not need to be considered, as the garbage collector will free memory no longer referenced.

On the other hand, F# can hold pointers to V8 JavaScript persistent handles (cf. §2.2). These handles need an explicit call to `v8::Dispose` or `v8::MakeWeak` in order to signal the garbage collector that the object can be deallocated. MiXture avoids this by having `JSValues` implement the `IDisposable` interface and override the `Finalize` method to call `MakeWeak` on the V8 persistent handle. F#’s garbage collector executes the destructor of `JSValues` when it is no longer accessible. V8 will then deallocate the object being referred to only if there are no other V8 persistent handles referencing the object.

The case for F# function values is more interesting: they share memory (the original function) since the strategy of embedding/projecting a function is to project/embed its arguments, run the original function, and embed/project the result value. The approach...
followed for de-allocating embedded F# functions is shown in Figure 3.9, and described following the example in the figure:

1. Function `make_list` is to be embedded, passing it to V8 as a function pointer using a delegate. This delegate is pinned using a `GCHandle` so that F#'s garbage collector doesn’t deallocate it.

2. The `GCHandle` is kept in a dictionary so that we can later retrieve it when freeing it.

3. When the F# pointer (`jmake_list_ptr`) is no longer accessible, `jmake_list_ptr.Finalize()` is called.

4. `jmake_list_ptr.Finalize()` makes the V8 persistent handle weak `jmake_list.MakeWeak()`.

5. If there are no other persistent handles to `jmake_list`, V8 de-allocates `jmake_list` and calls F# to free the `GCHandle` that held the delegate created in the embedding process.

![Figure 3.9: Memory management for embedded functions. The labels on top of the lines denote the embedding steps (in brackets) and go in the direction of the filled arrow heads. The labels under the lines denote the memory management process and go in the direction of empty arrow heads.](image)

3.3.9 Exception handling

Most implementations of multilanguage systems abort the entire execution of a program if an exception reaches a language boundary. While this is a valid design decision, the goal of this project is a very deep level of integration between F# and JavaScript to produce a more powerful system. Therefore, exceptions are translated if they ever reach the foreign
environment. The idea is simple: exceptions are caught at language boundaries and re-thrown on the other side of the boundary. In addition to notifying the occurrence of an exception, the values that the exception might carry are also translated. The following entities are defined:

- **JSException**, able to hold a **JSValue**, which points to the value being *thrown* from JavaScript, that can be projected.
- A JavaScript object with properties **name** (string “F# exception”) and **values** (array of values in the F# exception) is thrown when an F# exception reaches a FJ language boundary.

In conclusion, MiXture allows foreign exceptions to be dealt with using the native constructs of each language.

### 3.3.10 Convenient operators to deal with JavaScript values

Some operators are defined to perform some common tasks:

- Access object properties: **object name +> “property name”**. Having a **JSValue** value o pointing to a JavaScript object, a user can access property “p” from F# with o +> “p”.
- Function application: **function *@ argument list**. A function f is applied to arguments arg1, arg2, ..., argn by writing f *@ [arg1;arg2;...; argn]. This is desugared into exception handling code and a call to the C++ procedure applyFunctionArr.

### 3.3.11 Contexts and value registration

Recall from §2.2 that a V8 context is an execution environment with its own values defined. MiXture creates one on startup and sets it as the current one. The user can register values in the current context by providing a **JSValue** and the identifier string it should be associated with; this is done with register_values: ((string * JSValue) list -> unit). A user can also create a new execution context (create_context: unit -> JSValue) and set one as the current one (set_current_context: JSValue -> unit).

### 3.3.12 Polymorphism

*Polymorphism* is a programming language feature that allows values of different types to be handled using a uniform interface. Specifically, polymorphic functions can evaluate or
be applied to values of different types, thus having “multiple forms”.

There are several ways of implementing polymorphism, the simplest one being the use of duck typing: this allows a function to take parameters of different types as long as they provide some basic common properties. A more robust kind of polymorphism is parametric polymorphism, which uses type variables in place of ground types, which are then instantiated with particular types as needed. F# only supports let-polymorphism, disallowing functions that take polymorphic values as arguments [25 §22.7].

3.3.12.1 Embedding parametrically polymorphic F# functions

Basic polymorphism (one interface, different types) is automatically achieved when embedding an F# function because JavaScript is an untyped language (duck typing). This is, in fact, the approach chosen by Ramsey [5] and Benton [3] to allow the embedding of polymorphic functions. However, this technique allows the guest language to call a parametrically polymorphic F#/ML function in a non-safe manner.

This is illustrated in Listing 3.5 where the append function for arrays is embedded. Applying append to incompatible types (Number/int and string, in line 6 of the JavaScript session) is not restricted, and hence projecting the result of line 6 from the JavaScript transcript would cause a type error. The reason why append([0,1,2])(["hello","world"]) type checks and raises no exceptions in JavaScript is because F# reflection exposes parametrically polymorphic types with obj substituted for all type variables. Therefore, when Array.append is embedded in line 3 of the F# interactive session, we are actually embedding a value of type obj[] -> obj[] -> obj[].

The solution to this issue involves the use of contracts. Contracts ensure that the requirements of a function are never violated (i.e., the input values are in its domain, and the return values in its range). In the case of F# functions being embedded, contracts are needed only because F# exposes polymorphic types with obj, as otherwise F# is a type-safe language. For this reason, we unify the type variables of the actual function type being embedded with the types of the arguments provided by JavaScript. This involves the use of another F# meta-programming feature: quotations. Quotations provide a way to get a representation of F# expressions as abstract syntax trees, providing information about the actual (possibly parametrically polymorphic) type of expressions: for instance, getting the polymorphic type of Array.append and fst with the use of the implemented function create_signature is shown in Listing 3.6. The first element in the return tuple are the implicitly universally quantified type variables, the second element is the input type, and the third element is the return type. Therefore create_signature accurately determines

3Up to α-equivalence, in order to use the more common type variables α and β.
By using the active patterns implemented for a JavaScript handle, along with reflection, the unification is performed. If the unification succeeds, then the function application is performed, otherwise a PolymorphicEmbeddingException is raised. This safer embedding is performed by another function embed_poly_fun: Quotation.Expr -> JSValue (cf. Listing 3.7), as it requires a quoted expression to be its argument. Note that using embed_poly_fun to embed a polymorphic function, like embed, only prevents errors in F#, by ensuring that F# functions are not applied if their type constraints are not satisfied. This is done by the use of contracts and is shown in algorithm 3.3, where the intricate line 8 is the implementation of the rule $J$-to-$F$-$F$N from Figure 2.6.
3.3. NATURAL EMBEDDING

---

**F#**

```fsharp
> Array.append
val it : ('a [] -> 'a [] -> 'a []) = <fun:clo@62-2>
> register_values
   ["poly_append", embed_poly_func @@ Array.append @>]
val it : unit = ()
```

---

**JavaScript**

```javascript
> var append_naturals = poly_append([0,1,2])
function () { [native code] }
> append_naturals([3,4,5])
0,1,2,3,4,5
> append_naturals(['hello', 'world'])
Error: An F# exception occurred. Types not compatible
```

Listing 3.7: Interactive sessions transcript illustrating that using `embed_poly_func` is type-safe for F# polymorphic functions. Compare with Listing 3.5.

### 3.3.12.2 Projecting polymorphic JavaScript functions

As discussed above, JavaScript functions can be polymorphic because the language is untyped. However, ML-like languages such as F# have *value restriction* in their type systems, which limits the right-hand side of let-expressions to be syntactic values only when assigning to a polymorphic type value. As a result, we cannot produce a polymorphic function `poly_func` using `project<'T>`, since the right-hand side of the let-expression

```fsharp
let poly_func:'a->'b = project jsf // Value restriction error
```

wouldn’t be a syntactic term (for some type variables ’a, ’b, and a JSValue `jft` referencing a JavaScript function). We propose two solutions to circumvent this restriction:

- **η-conversion**, which corresponds to extensionality, i.e., \( (\lambda x. M x) \eta \leftrightarrow M \). As a result, we can employ

  ```fsharp
  > let jid = JSUtils.execute_string("(function(x) {return x;})")
  > let id x : 'a = (project jid) x // because id \eta \leftrightarrow project id
  val id : x: 'a -> 'a
  ```

- **type functions**, which take a type parameter as an argument (usually filled in with type inference). Thus we can use

  ```fsharp
  > let jid = JSUtils.execute_string("(function(x) {return x;})")
  > let id<'a> : 'a -> 'a = project jid
  val id<'a> : ('a -> 'a)
  ```
Algorithm 3.3 Embedding polymorphic functions.

Input:  
• `expr` is a quoted expression of a (possibly) polymorphic function.

1 function EMBED_POLY_FUN(expr)  
2    type_variables, domain, range, method ← CREATE_SIGNATURE(expr)  
3 function RESULT(js_arguments)  
4    input_types ← INFER_TYPES(args)  
5 switch UNIFY_TYPES(domain, input_types) do  
6    case Some(specialized_ty)  
7        projected_args ← PROJECT(js_arguments)  
8        EMBED(CALL_METHOD(method, specialized_ty, projected_args))  
9    case None  
10       ERROR(“Polymorphic function called with incompatible types”)  
11 return MAKE_JS_FUNCTION(result)

A sample use with type functions can be seen in Listing 3.8 where we define an array reversing function in JavaScript and project it to the type function `js_value` in lines 1–4. We then invoke the projected function for a list of float values and compare the result with the F# built-in reverse function for lists.

```
F#> let js_rev<'a> : 'a list -> 'a list =  
    "(function(arr) {return arr.reverse();})"  
 |> JSUtils.execute_string  
 |> project  
 val js_rev<'a> : ('a list -> 'a list)  
F#> List.rev  
 val it : ('a list -> 'a list) = <fun:clo@498>  
F#> let random_floats = [3.14; 2.71; 1.68]  
 val random_floats : float list = [3.14; 2.71; 1.68]  
F#> (List.rev random_floats) = (js_rev random_floats)  
 val it : bool = true
```

Listing 3.8: Projecting a JavaScript function into a polymorphic F# function.
3.4 Summary

This chapter described the implementation work carried out in this project. Both the lump embedding and the more powerful natural embedding have been implemented for the languages F# and JavaScript. The lump embedding did not prove to be as demanding to implement as the natural embedding. For the former, classes `a FSLump and JSLump were designed to represent the language boundaries. The implementation of the natural embedding is successful, being able to translate primitive values as well as an unlimited amount of function, collections and record types. Exception handling is also integrated in MiXture, allowing native constructs to deal with exceptions.

Finally, the work undertaken to ensure polymorphic functions are handled in a type-safe way was outlined, and this included the implementation of unification (embedding) and the use of $\eta$-conversion and type functions (projection).
Chapter 4

Evaluation

This chapter describes the systematic testing techniques used to verify the behavior of the system, and how the project was evaluated quantitatively and qualitatively.

Specifically, we measure the performance of MiXture by comparing execution times with the respective monolingual systems, and estimate the execution penalty of the translation of different data types. Finally, we present a formal quantitative study of MiXture by the use of cognitive dimensions.

4.1 Overall achievements

The success criteria from the project proposal (see Appendix E) were as follows (summarized and adapted for brevity):

Criteria 1 and 3. The lump and natural embedding implementations should be able to pass values from JavaScript to F# and then pass them back and make foreign calls. The natural embedding should in addition convert to native values.

These goals were achieved with the LEmbedding and NEmbedding modules described in §3, and were tested (module, functional and system testing), as outlined in §4.2.

Criterion 2. The resulting framework should not take significantly more time than executing the respective monolingual runtimes.

This was accomplished and evaluated with a series of standard benchmarks, and the results are reported in §4.3.1.

Criterion 4. A convenient syntax for multilanguage programming has been designed.

This objective was achieved by exposing a simple API. This criterion is assessed through a use case in §4.4, where we show the simpler syntax and rules of MiXtue in comparison to Lua-ML.
In addition to the above, we also perform benchmarking tests to assess the cost of embedding and projecting different data types \((\S 4.3.2)\), and how the cost changes with respect to the size of the value being embedded/projected \((\S 4.3.3)\).

4.2 Software testing

Testing software is recommended to be performed by a different team than the one which wrote the software library \([26]\). Due to the nature of this individual project, the author both researched testing techniques, and studied which were more suitable for this project.

4.2.1 Module (unit) testing

Module testing is a process of testing the individual units in a program. White-box module testing was performed for internal functions from the following modules (we provide detailed examples for \texttt{NEmbedding}):

- \texttt{LEmbedding};
- \texttt{NEmbedding}: several test cases for:
  - helper functions for \texttt{project} (\texttt{project_record}, \texttt{project_array}, \texttt{project_list}, etc.), with \texttt{Assert.AreEqual(\texttt{project (embed x)}, x)};
  - helper function for \texttt{embed} (\texttt{embed_ienumerable}, \texttt{string.embed}, \texttt{embed_poly_func}, etc.), with \texttt{Assert.IsTrue(JSUtils.strictCompare(\texttt{embed (project x)}, x))};
  - exceptions, with \texttt{Assert.Throws}.
- \texttt{JSUtils}.

The main benefits obtained from these tests were: (a) managed combined elements of testing; (b) eased debugging by re-running previously-completed tests (regression testing); and (c) allowed to test multiple modules simultaneously.

4.2.2 Functional testing: equivalence class partitioning

Functional testing is a black-box quality assurance process that attempts to find discrepancies between a program and the external specification. We use equivalence class partitioning
to reduce the total number of test cases necessary, while still providing a high degree of confidence of input coverage. This technique was employed to test the functions exposed by the interface files, by dividing the input into classes of valid and invalid sets. These two sets are further divided into equivalent subsets, whose elements provide equivalent test results: if one test case in an equivalence class does not detect an error, we expect that no other test cases in the equivalence class should fail (the elements in an equivalent class produce the “same” logical result [27, §5]).

The breadth of data coverage was increased by selecting random elements from a given subset to use as test data. This was performed by implementing random generators for the different test cases in the FsCheck [28] testing framework. Accordingly, each test case was run with 100 random values from each equivalence subset. The FsCheck generators were integrated with NUnit [29] to get visual reports in MonoDevelop — an example is shown in Figure 4.1.

<table>
<thead>
<tr>
<th>Input</th>
<th>Valid class subsets</th>
<th>vid</th>
<th>Invalid class subsets</th>
<th>iid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Positive float</td>
<td>v1</td>
<td>Not pointing to V8</td>
<td>i1</td>
</tr>
<tr>
<td></td>
<td>Negative float</td>
<td>v2</td>
<td>Integer greater than F# max</td>
<td>i2</td>
</tr>
<tr>
<td></td>
<td>String</td>
<td>v3</td>
<td>Integer less than F# min</td>
<td>i3</td>
</tr>
<tr>
<td></td>
<td>Positive integer</td>
<td>v4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Negative integer</td>
<td>v5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Array</td>
<td>v6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Function</td>
<td>v7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NaN</td>
<td>v8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>± Infinity</td>
<td>v9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Undefined</td>
<td>v10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Empty string</td>
<td>v11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Zero</td>
<td>v12</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Object</td>
<td>v13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>int</td>
<td>v14</td>
<td>Object type</td>
<td>i4</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>v15</td>
<td>Option type</td>
<td>i5</td>
</tr>
<tr>
<td></td>
<td>record</td>
<td>v16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unit</td>
<td>v17</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>string</td>
<td>v18</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>non-polymorphic function</td>
<td>v19</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bool</td>
<td>v20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>array</td>
<td>v21</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>list</td>
<td>v22</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: ECP table for project<’T>.
Here, we use the function `project<'T>` to illustrate the technique exercised. The equivalent sets for `project<'T>` are illustrated in Table 4.1: the two inputs of `project<'T>` are:

- the `JSValue` being projected (first row in the column “Input”),
- the output type parameter `'T` (second row in the column “Input”).

We can see in Figure 4.1 the test cases, which were obtained as follows:

1. Created a new test case covering as many of the uncovered valid equivalence classes as possible, until all valid equivalence classes were covered.
2. Wrote a test case that covered only one invalid equivalence class until all invalid equivalence classes were covered.

For the those requiring randomization, FsCheck generated 100 random values, but not for “singular” values test cases (e.g., ±∞).
4.2. SOFTWARE TESTING

4.2.3 System testing

System testing compares MiXture to its original objectives, and in this section we describe the two methods of system testing performed for this project.

4.2.3.1 Memory leaks: “heapshot analysis”

Memory leaks are a common problem for multilanguage systems, since either the user or the system must track two different environments of resources.

The Mono profiler was employed in heapshot analysis mode using statistical-based sampling to determine that pinned F# values were being freed. Figure 4.2 shows six heapshots of a program that runs in two loops, embedding a function 100,000 times in each loop. We force V8 GC to collect weak references after each loop. The different number of total instances of JSEngine.FSharpFunction (130,665) and those in the root set (29,336) in Figure 4.2c is explained by the fact that both garbage collectors are not synchronized. Nevertheless, we see in Figure 4.2d that the F# runtime also performs collections. The two JSEngine.FSharpFunction instances remaining at the end of test in Figure 4.2f are functions that MiXture provides to JavaScript: print and readline.

![Figure 4.2: Heapshot analysis results, sample number in parenthesis.](image)
4.2.3.2 Volume testing

Volume testing is a type of system testing that subjects the system to heavy volumes of data. We performed volume testing for both the number of values to be embedded and the size of the values. In the first case (§4.3.2), we determined that a maximum of \(6 \times 10^5\) functions was the limit. For the second case (§4.3.3): (a) the heap limit was reached for strings with length greater than \(4 \times 10^8\), because of the 2GB limit for .NET objects; and (b) the stack limit was reached for records with a recursion level greater than 53,000 (embed) and 3,500 (project). The limit for recursive records begs for more investigation, such as striving for a tail recursive implementation of `embed_record` and `project_record`.

Volume testing is tightly related to performance testing [27, §6] and hence we leave the specific methods and outcome for the next section.

4.3 Performance evaluation

Performance evaluation in Computer Science is sometimes not satisfactory (see [30] for some technical reports and evaluation anti-patterns), and hence in this dissertation we follow the advice of [30] and Le Boudec [31] to avoid some common anti-patterns, such as omitting measurement contexts, the statistical parameters (e.g., confidence level) or excluding outright data analysis (e.g., inferential statistics).

Benchmarking MiXtured gives an insight about the penalty incurred by using this framework for multilanguage programming, which is an important metric to decide whether MiXtured satisfies the requirements of the application to be implemented (success criterion 2).

For convenience and reproducibility, performance evaluation was automated using a Python script — `bench.py` — from *The Computer Language Benchmarks Game* [32]. With the use of a configuration file, this allowed us to easily modify the tests to be run and its parameters, and produced a CSV file as output that could be directly used by gnuplot to produce graphical representations of the data. All tests were performed on an Intel Core i5-3427U box running 32-bit Debian *Squeeze* GNU/Linux 2.6.32. Debian was chosen for its wide availability and standardization. Figure 4.3 shows a sample of tests being run.

All tests were run a number of times (specified in each section) and we report average CPU time (`usr+sys`). We use inferential statistics to establish the statistical significance of the results: confidence intervals with 95% confidence level following a Student’s \(t\)-distribution.
4.3. PERFORMANCE EVALUATION

4.3.1 Benchmarking MiXture against monolingual runtimes

Measurement bias involves measuring a new system in favorable conditions over the original system, with the intention of stating that its contributions are significant. We avoid it by using standard tests (binarytrees, fannkuchredux, fasta, mandelbrot, nbody, and spectralnorm) from [32] to compare the execution times using MiXture and the corresponding monolingual runtimes (Mono v2.10.9 [17] and V8 v3.15.10 [13]). Specifically, we report four results for each test: native runtimes, and multilanguage runtimes (MiXture), as illustrated in Table 4.2. Multilanguage test cases consist in executing the foreign source code, for which F# provides some standard functions to JavaScript such as print and readline. Each test was run 10 times.

<table>
<thead>
<tr>
<th>Source code</th>
<th>Runtime</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>V8</td>
<td>Native</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Mono</td>
<td>Using MiXture</td>
</tr>
<tr>
<td>F#</td>
<td>V8</td>
<td>Using MiXture</td>
</tr>
<tr>
<td>F#</td>
<td>Mono</td>
<td>Native</td>
</tr>
</tbody>
</table>

Table 4.2: Benchmarking for each test, associating source code and runtimes used.

The anticipated result was that the use of MiXture incurs a small performance penalty but not in a (non-statistically) significant way with respect to that of the programs being run.
in their native environments. Figure 4.4 reports average CPU times and the results are as expected. Program execution times did not vary by more than 2.3% (with an average of 1%) between the same source code and being run in different environments, which is an acceptable compromise for the ability of running a whole new language.

![Comparison between monolingual and multilanguage CPU execution times](image)

**Figure 4.4**: Benchmarking CPU execution time of monolingual and multilanguage systems. The input for each test is specified in brackets after each name.

Note that the startup time of Mono is considerable (it was estimated to be of 0.2 seconds), and has not been taken into account in order to reflect realistic execution times. Also note that error bars have not been included in Figure 4.4 because they are too small to discern graphically, the largest one being 1.4% (for “nobody in JavaScript”).

In order to have rigorous statistical results, we compute the confidence intervals for the difference between two means. In general, we expect the monolingual CPU execution time to be lower than that using MiXture, so we define

$$\mu = \mu_{\text{MiXture}} - \mu_{\text{monolingual}}$$
and expect the confidence intervals for $\mu$ to cross zero if the means are not significantly different, be positive if the MiXture times are significantly higher, and vice versa \[31, \S 2.2\].

We observe in Figure 4.5 that several confidence intervals do indeed cross the line $y = 0$, meaning that there is no significant difference between the execution times for MiXture and the native environments. As predicted, most of the other points lie above $y = 0$, implying that the execution time with MiXture is significantly higher, but they are generally close to $y = 0$ (and remember Mono’s start-up time has not been taken into account). Surprisingly, \texttt{fasta} and \texttt{nbody} ran faster when executing the JavaScript code in MiXture than in the native V8. This could be due to the amount of output generated to \texttt{stdout} by those two tests (note that the Python script used to run the tests and make measurements collects \texttt{stdout}, so the time printing to the terminal is appropriately ignored).

![Confidence intervals of difference of means for CPU execution times](image)

**Figure 4.5:** Difference of means of CPU execution times using MiXture and native runtimes.

These tests suggest that the difference in execution times must be due to:

- Mono startup time,
- conversions between datatypes,
- overhead in switching the runtime the control flow is assigned to, and
- generating output.
4.3.2 Comparing embedding and projection of different data types

In this section we explore the overhead incurred when embedding/projecting different data types. The expected result is that the primitive types (e.g., numerical types, ()/undefined) will have a very low penalty compared to more complex types such as records and functions. Due to the nature of the tests (project (embed x) in a for-loop), we anticipate that all results will be linear with the number of elements being used; a greater embed/project penalty will be seen as a steeper slope in the plots.

In order to conduct this part of the evaluation, values of each type being compared were embedded and subsequently projected (so it should act as the identity function); the results are described with respect to the number of times this identity operation was performed. The large number of conversions being performed resulted in lengthy tests (5 hours 48 minutes for the whole set), so all the tests were run 3 times for each “number of elements” being tested, and we report average execution times.

![Time versus number of elements being embedded and projected.](image.png)

**Figure 4.6** Performance tests for project/embed for different data types.

As we can see in Figure 4.6, our expectations are satisfied. Embedding and projecting the functions id and add are the most expensive conversions along with the simple record types website and int_record. On the other hand, primitive data types such as int/Number and ()/Undefined are inexpensive to translate. This is explained by the use of reflection in order to create a function / record type at runtime, according to the type inferred by

```fsharp
let id x = x
let add x y = x+y
type website =
    { Title : string;
      mutable Url : string }
type int_record =
    { I : int }
```
the compiler. For types such as float and int, most of the work is done via P/Invoke, hence its overhead is low. Among functions, the polymorphic function id was the most expensive to handle, and this is justified by the use of polymorphic contracts §3.3.12.

4.3.3 Benchmarking for the size of data types

In this section we look into the relationship between the size of a value and the CPU execution time required to embed/project it. It is not evident what the relationship is between the length of a string or the recursion level of a record and the time it takes to embed/project it. The tests were run 10 times and the average time is presented here.

4.3.3.1 Strings

Since strings have an underlying representation as arrays, we could expect both embed and project<string> to be linear in time with respect to the length of the string. For testing different string lengths (up to $2 \times 10^8$ characters, due to memory limitations), the string with $n$ “z” characters was embedded in one test, and also projected in another, where $n$ is the string length (x-axis in Figure 4.7).

![Time to embed and project strings versus string length](image)

Figure 4.7: Benchmarking CPU execution time versus the length of the string being embedded (blue) and projected (red).
In Figure 4.7, it appears that the trend for project<\text{string}> is higher than linear and about linear for embed. We test this hypothesis by plotting the same data in a log-log scale and estimating the linear line of best fit \( p(x) \), in which the coefficient for \( \ln(x) \) determines the exponent in the original plot. The coefficients are shown in Table 4.3 and are as expected from the raw data. The uncertainties are calculated by propagating them into the logarithms \( \ln(x) \) and estimating the maximum and minimum slopes. The time complexity for project<\text{string}> is above the anticipated value (linear); this may be due to the use of \texttt{strncpy} along with \texttt{StrinBuffers}. Overall, the results are satisfactory, and we can conclude that no obvious optimization is possible, as the results are close to linear.

<table>
<thead>
<tr>
<th>Test</th>
<th>Coefficient</th>
<th>( R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embed</td>
<td>0.94 ± 0.02</td>
<td>0.999</td>
</tr>
<tr>
<td>Project</td>
<td>1.23 ± 0.05</td>
<td>0.997</td>
</tr>
</tbody>
</table>

Table 4.3: Coefficient for the best linear line for \( \ln(x) \) in the log-log plot of Figure 4.7 (string size), along with the Pearson coefficient of correlation.

4.3.4 Recursive records

We now turn to study how MiXture behaves when handling records of increasing size. In order to programmatically create records of different size, we use the recursive record definitions in Listings 4.8c and 4.8d, which resemble an integer linked list, along with the creating functions \texttt{RL: int-> RList}.

Figures 4.8a and 4.8b display the results obtained, which follow a linear trend, with lines of best fit confirming this for Pearson coefficients of correlation of 0.999 and 0.994 for embed<\texttt{RList}> and project<\texttt{RList}>, respectively. Even though these results seem to confirm the linearity of embed and project for record types, we acknowledge the need for a wider range of recursion levels, which was not possible due to memory constraints (§4.2.3.2).

\[^{1}\Delta (\ln y) = \frac{\Delta y}{y}\]
4.3. PERFORMANCE EVALUATION

- **Time to embed records versus record recursion level**
  - (a) Embed.

- **Time to project records versus record recursion level**
  - (b) Project.

The graphs show the benchmarking CPU execution time versus the record recursion level for both embedding and projecting records.

**Definitions for Figure 4.8a (F#)**

```fsharp
type RList = { head : int;  tail : RList }
let rec RL n =  if n = 0 then    Unchecked.defaultof<_>  else    { head = n;  tail = RL (n - 1) }
```

**Definitions for Figure 4.8b (JavaScript)**

```javascript
function RL(n) {  if (n==0) {    return undefined;  }  else {    return { head : n,    tail : RL(n-1)};  }}
```

Figure 4.8: Benchmarking CPU execution time versus the record recursion level.
4.4 Qualitative evaluation: cognitive dimensions

This section addresses success criterion 4 (convenient syntax). For this purpose, we use the cognitive dimensions (CDs) framework \[33, \S 5\], in which each dimension describes an aspect of an information structure.

Here, we use the similar system Lua-ML and compare it with MiXture according to the CDs framework. We present the results and conclusions drawn in Table 4.4. The full discussion can be found in Appendix C.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>MiXture</th>
<th>Lua-ML</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Viscosity</td>
<td>✓</td>
<td>×</td>
<td>High-levels viscosity break the train of thought and might cause low productivity in a developer when using two programming languages simultaneously</td>
</tr>
<tr>
<td>Visibility</td>
<td>→ ✓</td>
<td></td>
<td>MiXture supports similar visibility, but doesn’t require the user to deal with it</td>
</tr>
<tr>
<td>Diffuseness</td>
<td>✓</td>
<td>×</td>
<td>Trade-off with visibility</td>
</tr>
<tr>
<td>Consistency</td>
<td>✓</td>
<td>×</td>
<td>Native vs non-native type annotations; polymorphism</td>
</tr>
</tbody>
</table>

Table 4.4: Cognitive dimensions study summary.

4.5 Summary

The evaluation of this project has been described in this chapter from several perspectives, including a) software testing for correctness, and b) several kinds of performance benchmarking to establish that MiXture imposes no (or barely) statistically significant penalty, as well as measure the effect performing a large number of translations.

We performed a qualitative evaluation using cognitive dimensions. We concluded that MiXture has low viscosity (✓), average-low visibility (×), low diffuseness (✓) and high consistency (✓).
Chapter 5

Conclusions

This dissertation describes the research, design, implementation, and evaluation carried out in order to implement MiXture, a multilanguage framework for the .NET language F#, and JavaScript. We conclude by considering further extensions to this framework that would be required to achieve a deeper level of interoperability (§5.1), and discussing the achievements of the project (§5.2).

5.1 Future work

There are a number of ways this project could be extended. An outline of possible extensions and the challenges they pose is presented next.

5.1.1 F# objects

This dissertation has not addressed .NET objects from F#, because the size of the project would have increased considerably, and because of the very unclear mapping that these values would have. It is likely that not supporting .NET objects outright was a good decision, as it would compromise unnecessarily when handling them.

5.1.2 Other polymorphic data types

In §3.3.12 we outlined the problem with handling parametrically polymorphic F# functions (both embedding and projection). There are, however, other functional data types that are commonly used in F# such as union types and polymorphic records.

The main challenge with union types is that there is no clear equivalent in native JavaScript. Nevertheless, this could be implemented by providing a JavaScript datastructure matching
the interface of union types.

The task with polymorphic records (and union types) is the value restriction when projecting. For polymorphic functions, we evaded this restriction by using $\eta$-conversion and type functions, but these are not appropriate for records and unions. A new strategy would need to be devised in order to address this extension.

5.1.3 Multi-threaded applications

JavaScript, unlike F#, does not support multithreading, and bases asynchronous programming on the event-driven paradigm. An avenue for further work might be to combine asynchronous programs in an integrated fashion. It might be worth using the work implemented for Scala by Haller and Odersky [34], in which they introduce a unification of thread-based and event-based actors by avoiding inversion of control \footnote{the Hollywood Principle: “Don’t call us, we’ll call you”} in the latter, and allowing actors to wait on messages using two different operations: one corresponding to thread-based programming and the other to event-based programming.

5.2 Accomplishments

MiXture is an F# library that implements the lump and natural embedding for F# and JavaScript. A library-based implementation requires neither special syntax nor compiler support, so MiXture can be easily extended. Installation is also made easier, as all that is required is a single DLL drop-in.

The library-implementation approach allowed us to write high-level code, so we could implement a large number of features that are novel from the point of view of research in programming languages. For a mixed environment of languages while still preserving type-safety, these features include collections, parametric polymorphism, memory management, exception handling, etc.

The result of this project provides an embedding API where type-safety is guaranteed: a runtime error is not possible due to an error in glue code. To achieve this, a considerable amount of research was carried out, from the original lump and natural embeddings, to the use of parametric contracts in order to ensure type-safety for polymorphic functions.

The evaluation performed in §4 reassures us that the implementation is correct with respect to the intended semantics and behaves in a stably when operated outside standard usage.
Bibliography


[29] NUnit. [http://www.nunit.org/]


Appendix A

Type-safety proofs

We prove type-safety of $\mathcal{F}$ by the standard combination of type preservation and progress. In this appendix, we provide the cases added to the natural embedding theory (i.e., functions, records, lists, tuples, arrays). The other general cases can be found in [1].

We first specify the calculi we work with. $\mathcal{F}$ is a stand-in for F#, and $\mathcal{J}$ is a stand-in for JavaScript.
A.1 \( \mathcal{F} \) collected definition

Syntax
\[
e ::= x \mid v \mid (e \ e) \mid \text{head} \ e \mid \text{tail} \ e \quad v ::= \lambda x : \tau. \ e \mid \text{true} \mid \text{false} \mid (v, v) \mid \text{nil}
\]
\[
| \{\text{lab}_1 \ = \ e; \ldots ; \text{lab}_k \ = \ e\} \mid \#\text{lab} \ e
\]
\[
e ::= e \mid (e, e) \mid \#1 \ e \mid \#2 \ e \quad x \ \overset{\text{def}}{=} \text{variables in} \ \mathcal{F}
\]
\[
\mathcal{E} ::= [\ ]_\mathcal{F} \mid \mathcal{E} \ e \mid v \ \mathcal{E} \mid \#1 \ \mathcal{E} \mid \#2 \ \mathcal{E}
\]
\[
| \{\text{lab}_1 \ = \ e; \ldots ; \text{lab}_i \ = \ \mathcal{E}; \ldots ; \text{lab}_k \ = \ e\}
\]
\[
| \#\text{lab} \ \mathcal{E} \mid \text{head} \ \mathcal{E} \mid \text{tail} \ \mathcal{E} \mid v :: \mathcal{E} \mid \mathcal{E} :: e
\]

Typing rules
\[
\begin{align*}
\text{(BOOL)} & \quad \Gamma \vdash b : \text{bool}, \text{if } b \in \{\text{true}, \text{false}\} \\
\text{(FN)} & \quad \Gamma, x : \tau_1 \vdash e : \tau_2 \\
\text{(REC)} & \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_k : \tau_k \\
\text{(RECP)} & \quad \Gamma \vdash \{\text{lab}_1 = e_1; \ldots ; \text{lab}_k = e_k\} : \{\text{lab}_1 : \tau_1; \ldots ; \text{lab}_k : \tau_k\}
\end{align*}
\]
\[
\begin{align*}
\text{(PAIR)} & \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\text{(CONS)} & \quad \Gamma, e_1 : \tau \vdash e_2 : \tau \ \text{list} \\
\text{(NIL)} & \quad \Gamma \vdash \text{nil} : \tau \ \text{list}
\end{align*}
\]
\[
\begin{align*}
\text{(HEAD)} & \quad \Gamma \vdash e : \tau \ \text{list} \\
\text{(TAIL)} & \quad \Gamma \vdash \text{head} \ e : \tau \\
\end{align*}
\]

Operational semantics
\[
\begin{align*}
\text{(FN)} & \quad \mathcal{E}[\lambda x : \tau. \ e] \ v \rightarrow \mathcal{E}[e [v/x]] \\
\text{(REC)} & \quad \mathcal{E}[\#\text{lab}_i \ \{\text{lab}_1 = v_1; \ldots ; \text{lab}_k = v_k\}] \rightarrow \mathcal{E}[v_i]
\end{align*}
\]
\[
\begin{align*}
\text{(PAIR1)} & \quad \mathcal{E}[\#1(v_1, v_2)] \rightarrow \mathcal{E}[v_1] \\
\text{(PAIR2)} & \quad \mathcal{E}[\#2(v_1, v_2)] \rightarrow \mathcal{E}[v_2]
\end{align*}
\]
\[
\begin{align*}
\text{(HEAD)} & \quad \mathcal{E}[\text{head} \ (v_1 :: v_2)] \rightarrow \mathcal{E}[v_1] \\
\text{(TAIL)} & \quad \mathcal{E}[\text{tail} \ (v_1 :: v_2)] \rightarrow \mathcal{E}[v_2]
\end{align*}
\]
A.2 \( J \) collected definition

Syntax

\[
e ::= x | v | \text{head } e | \text{tail } e \\
| \{lab_1 = e; \ldots ; lab_k = e\} | \#lab e \\
| \text{nil} | e :: e
\]

\[
v ::= \lambda x. e | \text{true} | \text{false} | \text{nil}
\]

\[
x \overset{\text{def}}{=} \text{variables in } J
\]

\[
E ::= [ ]_J | E e | v E \\
| \{lab_1 = e; \ldots ; lab_i = E; \ldots ; lab_k = e\} \\
| \#lab E | \text{head } E | \text{tail } E | v :: E | E :: e
\]

Typing rules

\[
\frac{}{\Gamma \vdash e : JT} \quad \text{(All)}
\]

Operational semantics

\[
\frac{}{E[(\lambda x. e) v] \rightarrow E[e[v/x]]} \quad \text{(FN)}
\]

\[
\frac{}{E[\#lab_i \{lab_1 = v_1; \ldots ; lab_k = v_k\}] \rightarrow E[v_i]} \quad \text{(REC)}
\]

\[
\frac{}{E[\text{head } (v_1 :: v_2)] \rightarrow E[v_1]} \quad \text{(HEAD)}
\]

\[
\frac{}{E[\text{tail } (v_1 :: v_2)] \rightarrow E[v_2]} \quad \text{(TAIL)}
\]
A.3 Natural embedding

The natural embedding has been extended in this dissertation in order to cope with lists, collections, etc. The reader is referred to [1, §3.5] for a formal discussion of exceptions, which has been omitted here due to space constraints.

Syntax

\[
\begin{align*}
\text{e} & ::= \cdots | FJG^\tau e \\
\text{E} & ::= \cdots | FJG^\tau E
\end{align*}
\]

Typing rules

\[
(\mathcal{F}\text{-TRANS}) \quad \frac{}{\Gamma \vdash FJG^\tau e : \tau}
\]

\[
(\mathcal{J}\text{-TRANS}) \quad \frac{}{\Gamma \vdash GJF^\tau e : J\mathcal{T}}
\]

Operational semantics

\[
(\mathcal{J}\text{-to-}\mathcal{F}\text{-BOOL}) \quad \frac{}{\mathcal{E}[FJG^{\text{bool}}(b)] \rightarrow \mathcal{E}[b]}
\]

\[
(\mathcal{F}\text{-to-}\mathcal{J}\text{-BOOL}) \quad \frac{}{\mathcal{E}[GJF^{\text{bool}}(b)] \rightarrow \mathcal{E}[b]}
\]

where \((b, b) \in \{ (\text{true, true}), (\text{false, false}) \} \)

\[
(\mathcal{J}\text{-to-}\mathcal{F}\text{-B-ERROR}) \quad \frac{}{\mathcal{E}[FJG^{\text{bool}}(v)] \rightarrow \mathcal{E}[FJG^{\text{bool}}(\text{error("not-bool")})]}
\]

if \(v \notin \{ \text{true, false} \}\)

\[
(\mathcal{J}\text{-to-}\mathcal{F}\text{-FN}) \quad \frac{}{\mathcal{E}[FJG^{\tau_1 \rightarrow \tau_2}(\lambda x. e)] \rightarrow \mathcal{E}[\lambda x : \tau_1. FJG^{\tau_2}((\lambda x : \tau_1. e)(GJF^{\tau_1}x))]}
\]

\[
(\mathcal{F}\text{-to-}\mathcal{J}\text{-FN}) \quad \frac{}{\mathcal{E}[GJF^{\tau_1 \rightarrow \tau_2}(\lambda x : \tau_1. e)] \rightarrow \mathcal{E}[\lambda x. GJF^{\tau_2}((\lambda x : \tau_1. e)(FJG^{\tau_1}x))]}
\]

\[
(\mathcal{J}\text{-to-}\mathcal{F}\text{-FN-ERROR}) \quad \frac{}{\mathcal{E}[FJG^{\tau_1 \rightarrow \tau_2}(v)] \rightarrow \mathcal{E}[FJG^{\tau_1 \rightarrow \tau_2}(\text{error("not-fn")})]}
\]

if \(v \neq \lambda x. e\), for any \(x\) or \(e\).
A.3. NATURAL EMBEDDING

\[(\mathcal{J}\text{-to-}\mathcal{F}\text{-Rec})\]

\[
\mathcal{E}[FJG^{\{lab_1:\tau_1;\ldots;lab_k:\tau_k\}}(\{lab_1 = v_1; \ldots; lab_k = v_k\})] \rightarrow \\
\mathcal{E}[\{lab_1 = FJG^{\tau_1}(v_1); \ldots; lab_k = FJG^{\tau_k}(v_k)\}]
\]

\[(\mathcal{F}\text{-to-}\mathcal{J}\text{-Rec})\]

\[
\mathcal{E}[GJF^{\{lab_1:\tau_1;\ldots;lab_k:\tau_k\}}(\{lab_1 = v_1; \ldots; lab_k = v_k\})] \rightarrow \\
\mathcal{E}[\{lab_1 = FJF^{\tau_1}(v_1); \ldots; lab_k = GJF^{\tau_k}(v_k)\}]
\]

\[(\mathcal{J}\text{-to-}\mathcal{F}\text{-R-error})\]

\[
\mathcal{E}[FJG^{\{lab_1:\tau_1;\ldots;lab_k:\tau_k\}}(v)] \rightarrow \mathcal{E}[FJG^{\{lab_1:\tau_1;\ldots;lab_k:\tau_k\}}(\text{error}(\text{“not-rec”}))]
\]

if \(v \neq (\{lab_1 = v_1; \ldots; lab_k = v_k\})\), for any \(v_i\) (1 \(\leq\) \(i\) \(\leq\) \(k\)) such that \(\mathcal{E}[FJG^{\tau_i}(v_i)] \not\rightarrow \mathcal{E}[\text{error}].\)

\[(\mathcal{J}\text{-to-}\mathcal{F}\text{-List})\]

\[
\mathcal{E}[FJG^{\tau\text{ list}}(v_1 :: v_2)] \rightarrow \mathcal{E}[(FJG^{\tau}(v_1)) :: (FJG^{\tau\text{ list}}(v_2))]
\]

\[(\mathcal{F}\text{-to-}\mathcal{J}\text{-List})\]

\[
\mathcal{E}[GJF^{\tau\text{ list}}(v_1 :: v_2)] \rightarrow \mathcal{E}[(FJG^{\tau}(v_1)) :: (FJG^{\tau\text{ list}}(v_2))]
\]

\[(\mathcal{J}\text{-to-}\mathcal{F}\text{-L-error})\]

\[
\mathcal{E}[FJG^{\tau\text{ list}}(v)] \rightarrow \mathcal{E}[FJG^{\tau\text{ list}}(\text{error}(\text{“not-list”}))]
\]

if \(v \neq (v_1 :: v_2)\), for any \(v_1, v_2\) such that \(\mathcal{E}[FJG^{\tau}(v_1)] \not\rightarrow \mathcal{E}[\text{error}]\) and \(\mathcal{E}[FJG^{\tau\text{ list}}(v_2)] \not\rightarrow \mathcal{E}[\text{error}].\)

\[(\mathcal{J}\text{-to-}\mathcal{F}\text{-Tuple})\]

\[
\mathcal{E}[FJG^{\tau_1\ldots\tau_k}(v_1 :: \ldots :: v_k :: \text{nil})] \rightarrow \mathcal{E}[(FJG^{\tau_1}(v_1), \ldots, (FJG^{\tau_k}(v_k)))]
\]

\[(\mathcal{F}\text{-to-}\mathcal{J}\text{-Tuple})\]

\[
\mathcal{E}[GJF^{\tau_1\ldots\tau_k}((v_1, \ldots, v_k))] \rightarrow \mathcal{E}[(FJG^{\tau_1}(v_1), \ldots, (FJG^{\tau_k}(v_k)) :: \text{nil}]
\]

\[(\mathcal{J}\text{-to-}\mathcal{F}\text{-T-error})\]

\[
\mathcal{E}[FJG^{\tau_1\ldots\tau_k}(v)] \rightarrow \mathcal{E}[FJG^{\tau_1\ldots\tau_k}(\text{error}(\text{“not-tuple”}))]
\]

if \(v \neq (v_1, \ldots, v_k)\), for any \(v_i\) (1 \(\leq\) \(i\) \(\leq\) \(k\)) such that \(\mathcal{E}[FJG^{\tau_i}(v_i)] \not\rightarrow \mathcal{E}[\text{error}].\)
APPENDIX A. TYPE-SAFETY PROOFS

A.4 Proofs

The cases handled in this section are original proofs for the extensions performed to the natural embedding. We use the standard proof technique by Wright and Felleisen [35] to prove type safety, first proving type preservation (§A.4.1) and progress (§A.4.2).

A.4.1 Type preservation

**Theorem** (Type preservation). If $\Gamma \vdash e : \tau$ and $\mathcal{E}[e] \rightarrow \mathcal{E}[e']$ then $\Gamma \vdash e' : \tau$.

**Proof.** We prove type preservation by rule induction on reduction derivations $\mathcal{E}[e] \rightarrow \mathcal{E}[e']$

A.4.1.1 $J$-to-$\mathcal{F}$ cases

**Case** ($J$-to-$\mathcal{F}$-FN).

\[ \frac{\mathcal{E}[FJG^{\tau_1 \rightarrow \tau_2}(\lambda x. e)] \rightarrow \mathcal{E}[\lambda x : \tau_1. FJG^{\tau_2}(\lambda x. e)(GJF^{\tau_1}x)]]}{\mathcal{E}[FJG^{\tau_1 \rightarrow \tau_2}(\lambda x. e)] \rightarrow \mathcal{E}[\lambda x : \tau_1. FJG^{\tau_2}(\lambda x. e)(GJF^{\tau_1}x)]]} \]

Assume

\[ \Gamma \vdash FJG^{\tau_1 \rightarrow \tau_2}(\lambda x. e) : \tau \]

The last rule in the typing derivation must have been ($\mathcal{F}$-TRANS), and hence $\tau = \tau_1 \rightarrow \tau_2$

Considering the reduction of $FJG^{\tau_1 \rightarrow \tau_2}(\lambda x. e)$ according to ($J$-to-$\mathcal{F}$-FN), by ($\mathcal{F}$-TRANS) we have

\[ \Gamma, x : \tau_1 \vdash FJG^{\tau_2}(\lambda x. e)(GJF^{\tau_1}x) : \tau_2 \]

We can then use (FN) to derive

\[ \Gamma \vdash \lambda x : \tau_1. FJG^{\tau_2}(\lambda x. e)(GJF^{\tau_1}x) : \tau_1 \rightarrow \tau_2 \]

as required.

**Case** ($J$-to-$\mathcal{F}$-Rec).

\[ \frac{\mathcal{E}[FJG^{\{lab_1:\tau_1; \ldots; lab_k:\tau_k\}}\{\{lab_1 = v_1; \ldots; lab_k = v_k\}\}]} \rightarrow \mathcal{E}[\{lab_1 = FJG^{\tau_1}(v_1); \ldots; lab_k = FJG^{\tau_2}(v_k)\}]}{\mathcal{E}[\{lab_1 = FJG^{\tau_1}(v_1); \ldots; lab_k = FJG^{\tau_2}(v_k)\]}} \]

Assume

\[ \Gamma \vdash FJG^{\{lab_1:\tau_1; \ldots; lab_k:\tau_k\}}\{\{lab_1 = v_1; \ldots; lab_k = v_k\} : \tau \]
The last rule in the typing derivation must have been ($\mathcal{F}$-Trans), and hence
\[ \tau = \{ \text{lab}_1 : \tau_1; \ldots ; \text{lab}_k : \tau_k \}. \]

Considering the reduction of $FJG^{\{\text{lab}_1 = v_1; \ldots ; \text{lab}_k = v_k\}}$ according to ($\mathcal{J}$-TO-$\mathcal{F}$-Rec), by ($\mathcal{F}$-Trans) we have
\[
\begin{align*}
\Gamma \vdash (FJG^{\tau_1}(v_1)) : \tau_1 \quad &\text{(A.1)} \\
\vdots \quad &\text{(A.2)} \\
\Gamma \vdash (FJG^{\tau_k}(v_k)) : \tau_k \quad &\text{(A.3)}
\end{align*}
\]

We can then use (A.1), (A.2) and (A.3) with (Rec) to derive
\[ \Gamma \vdash \{ \text{lab}_1 = FJG^{\tau_1}(v_1); \ldots ; \text{lab}_k = FJG^{\tau_k}(v_k) \} : \{ \text{lab}_1 : \tau_1; \ldots ; \text{lab}_k : \tau_k \} \]
as required.

**Case** ($\mathcal{J}$-TO-$\mathcal{F}$-List).
\[
\mathcal{E}[FJG^{\tau}_{\text{list}}(v_1 :: v_2)] \rightarrow \mathcal{E}[(FJG^{\tau_1}(v_1)) :: (FJG^{\tau_\text{list}}(v_2))] \]

Assume
\[ \Gamma \vdash FJG^{\tau'}_{\text{list}}(v_1 :: v_2) : \tau \]

The last rule in the typing derivation must have been ($\mathcal{F}$-Trans), and hence $\tau = \tau' \text{ list}$. Considering the reduction of $FJG^{\tau'_\text{list}}(v_1 :: v_2)$ according to ($\mathcal{J}$-TO-$\mathcal{F}$-List), by ($\mathcal{F}$-Trans) we have
\[
\begin{align*}
\Gamma \vdash (FJG^{\tau'}(v_1)) : \tau' \quad &\text{(A.4)} \\
\Gamma \vdash (FJG^{\tau'_{\text{list}}}(v_2)) : \tau' \text{ list} \quad &\text{(A.5)}
\end{align*}
\]

We can then use (A.4) and (A.5) with (Cons) to derive
\[ \Gamma \vdash (FJG^{\tau'}(v_1)) :: (FJG^{\tau'_{\text{list}}}(v_2)) : \tau' \text{ list} \]
as required.

**Case** ($\mathcal{J}$-TO-$\mathcal{F}$-Tuple).
\[
\mathcal{E}[FJG^{\tau_1 \ast \ldots \ast \tau_k}(v_1 :: \ldots :: v_k :: \text{nil})] \rightarrow \mathcal{E}[(FJG^{\tau_1}(v_1)), \ldots , (FJG^{\tau_k}(v_k))] \]

Assume
\[ \Gamma \vdash FJG^{\tau_1 \ast \ldots \ast \tau_k}(v_1 :: \ldots :: v_k :: \text{nil}) : \tau \]
The last rule in the typing derivation must have been \((F\text{-Trans})\), and hence \(\tau = \tau_1 * \ldots * \tau_k\).

Considering the reduction of \(FJG^{\tau_1 * \ldots * \tau_k}(v_1 :: \ldots :: v_k :: nil)\) according to \((J\text{-to-}F\text{-Tuple})\), by \((F\text{-Trans})\) we have

\[
\Gamma \vdash (FJG^{\tau_1}(v_1)) : \tau_1
\]
\[
\vdots
\]
\[
\Gamma \vdash (FJG^{\tau_k}(v_k)) : \tau_k
\]

We can then use (A.6), (A.7) and (A.8) with \((\text{Pair})\) to derive

\[
\Gamma \vdash ((FJG^{\tau_1}(v_1)), \ldots, (FJG^{\tau_k}(v_k))) : \tau_1 * \ldots * \tau_k
\]
as required.

### A.4.1.2 \(F\text{-to-}J\) cases

The cases in the other direction are fairly trivial, as we need only apply \((F\text{-Trans})\).

**Case** \((F\text{-to-}J\text{-Rec})\).

\[
\frac{\mathcal{E}[GJF^{\{lab_1 : \tau_1; \ldots; lab_k : \tau_k\}}(\{lab_1 = v_1; \ldots; lab_k = v_k\})] \rightarrow \mathcal{E}[\{lab_1 = FJF^{\tau_1}(v_1); \ldots; lab_k = GJF^{\tau_k}(v_k)\}]}
\]

Assume

\[
\Gamma \vdash FJG^{\tau}(GJF^{\{lab_1 : \tau_1; \ldots; lab_k : \tau_k\}}(\{lab_1 = v_1; \ldots; lab_k = v_k\})) : \tau
\]

Considering the reduction of \(FJG^{\tau}(GJF^{\{lab_1 : \tau_1; \ldots; lab_k : \tau_k\}}(\{lab_1 = v_1; \ldots; lab_k = v_k\}))\) according to \((F\text{-to-}J\text{-Rec})\), by \((F\text{-Trans})\) we have

\[
\Gamma \vdash FJG^{\tau}(\{lab_1 = FJF^{\tau_1}(v_1); \ldots; lab_k = GJF^{\tau_k}(v_k)\}) : \tau
\]
as required.

**Case** \((F\text{-to-}J\text{-List})\).

\[
\frac{\mathcal{E}[GJF^{\tau} \text{list}(v_1 :: v_2)] \rightarrow \mathcal{E}[(FJG^{\tau}(v_1)) :: (FJG^{\tau} \text{list}(v_2))]}{}
\]

Assume

\[
\Gamma \vdash FJG^{\tau}(GJF^{\tau} \text{list}(v_1 :: v_2)) : \tau
\]

Considering the reduction of \(FJG^{\tau}(GJF^{\tau} \text{list}(v_1 :: v_2))\) according to \((F\text{-to-}J\text{-List})\), by \((F\text{-Trans})\) we have

\[
\Gamma \vdash FJG^{\tau}((FJG^{\tau'}(v_1)) :: (FJG^{\tau'} \text{list}(v_2))) : \tau
\]
as required.
A.4. PROOFS

Case (F-to-J-Tuple).

\[ \mathcal{E}[GJF^{\tau_1 \times \ldots \times \tau_k}((v_1, \ldots, v_k))] \to \mathcal{E}[(FJG^{\tau_1}(v_1)) :: \ldots :: (FJG^{\tau_k}(v_k)) :: \text{nil}] \]

Assume

\[ \Gamma \vdash FJG^{\tau}(GJF^{\tau_1 \times \ldots \times \tau_k}((v_1, \ldots, v_k))) : \tau \]

Considering the reduction of \( FJG^{\tau}(GJF^{\tau_1 \times \ldots \times \tau_k}((v_1, \ldots, v_k))) \) according to (F-to-J-Tuple), by (F-Trans) we have

\[ \Gamma \vdash FJG^{\tau}((FJG^{\tau_1}(v_1)) :: \ldots :: (FJG^{\tau_k}(v_k)) :: \text{nil}) : \tau \]

as required.

A.4.1.3 J’s error cases

The cases

(F-to-J-B-error)
(F-to-J-FN-error)
(F-to-J-R-error)
(F-to-J-L-error)
(F-to-J-T-error)

are rather trivial, as they all make an equivalent erroneous step, after which the typing rule (F-Trans) can be applied without further complications. We show the case for (F-to-J-R-error), as the others are similar.

Case (J-to-J-R-error).

\[ \mathcal{E}[FJG^{\{lab_1 : \tau_1; \ldots ; lab_k : \tau_k\}}(v)] \to \mathcal{E}[FJG^{\{lab_1 : \tau_1; \ldots ; lab_k : \tau_k\}}(\text{error("not-rec")})] \]

if \( v \neq (\{lab_1 = v_1; \ldots ; lab_k = v_k\}) \), for any \( v_i \) (1 \( \leq i \leq k \)) such that

\[ \mathcal{E}[FJG^{\tau_i}(v_i)] \not\to \mathcal{E}[\text{error}] \]

Assume

\[ \Gamma \vdash FJG^{\{lab_1 : \tau_1; \ldots ; lab_k : \tau_k\}}(v) : \{lab_1 : \tau_1; \ldots ; lab_k : \tau_k\} \]

Considering the reduction of \( FJG^{\{lab_1 : \tau_1; \ldots ; lab_k : \tau_k\}}(v) \) according to (J-to-J-R-error), by (F-Trans) we have

\[ \Gamma \vdash FJG^{\{lab_1 : \tau_1; \ldots ; lab_k : \tau_k\}}(\text{error("not-rec")}) : \{lab_1 : \tau_1; \ldots ; lab_k : \tau_k\} \]

as required.
A.4.2 Progress

Theorem (Progress). If $\Gamma \vdash e : \tau$ then either $e$ is a value or, for all evaluation contexts $E$, there exists an $e'$ such that $E[e']$ or $E[\text{error}]$ ($J$ is the only source of errors).

Proof. Progress is proved by rule induction on type derivations. We haven’t introduced any original typing rules for the natural embedding, so we don’t reproduce the proof for $(F\text{-TRANS})$, which is described in [1, §3.2]. The other standard typing rules of $F$ (e.g., Rec, Cons, etc.) can be found in [25, §11].

A.4.3 Type-safety

We finally arrive at type-safety:

Theorem (Type-safety for $F$). A well-typed expression $e$ in $F$, $e \vdash \tau$, doesn’t get “stuck”: either $E[e] \rightarrow E[v], E[e] \rightarrow E[\text{error}]$ ($J$ is the only source of errors), or $e$ diverges.

Proof. Using the standard proof technique for safety by Wright and Felleisen [35], theorems (Type preservation) and (Progress) determine type-safety property of calculus $F$, when used in the natural embedding with $J$. \qed
Appendix B

Translation rules

The following sections (§B.1–§B.4) show the formal semantics of the translation rules as implemented in MiXture. Note that the subscripts $_{JS}$ and $_{F#}$ indicate the language a value belongs to. So $s_{JS}$ is the F# string $s_{F#}$ in JavaScript. These are all original translation rules developed by the author.

Table B.1 shows the corresponding types as MiXture supports them.

<table>
<thead>
<tr>
<th>F# type</th>
<th>JavaScript type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Number</td>
</tr>
<tr>
<td>float</td>
<td>Number</td>
</tr>
<tr>
<td>string</td>
<td>String</td>
</tr>
<tr>
<td>bool</td>
<td>Boolean</td>
</tr>
<tr>
<td>()</td>
<td>Undefined</td>
</tr>
<tr>
<td>null (any type that supports it)</td>
<td>Null</td>
</tr>
<tr>
<td>${lab_1: \tau_1; \ldots; lab_k: \tau_k}$</td>
<td>Object</td>
</tr>
<tr>
<td>$\tau$ list list</td>
<td>Array</td>
</tr>
<tr>
<td>$\tau_1 \star \ldots \star \tau_k$</td>
<td>Array</td>
</tr>
<tr>
<td>$\tau$ array</td>
<td>Array</td>
</tr>
<tr>
<td>exn</td>
<td>Object</td>
</tr>
<tr>
<td>$\tau_1 \rightarrow \tau_2$</td>
<td>Function</td>
</tr>
</tbody>
</table>

Table B.1: Type mappings for MiXture.
B.1 Primitives

\( (\text{Proj-Int}) \)
\[
    f_{\text{JS}} \quad \xrightarrow{\text{project}} \quad n_{\text{F#}} \\
    \text{if } \text{truncate}(f_{\text{JS}}) = f_{\text{JS}} \text{ and } n_{\text{F#}} = \text{truncate}(f_{\text{JS}})
\]

\( (\text{Proj-Int-Err}) \)
\[
    f_{\text{JS}} \quad \xrightarrow{\text{project}} \quad \text{Error} \\
    \text{if } \text{truncate}(f_{\text{JS}}) \neq f_{\text{JS}}
\]

\( (\text{Proj-Float}) \)
\[
    f_{\text{JS}} \quad \xrightarrow{\text{project}} \quad f_{\text{F#}} \\
    \text{where } f \text{ is a floating-point number}
\]

\( (\text{Embed-String}) \)
\[
    s_{\text{JS}} \quad \xrightarrow{\text{embed}} \quad s_{\text{F#}} \\
    \text{where } s \text{ is a string}
\]

\( (\text{Proj-String}) \)
\[
    s_{\text{JS}} \quad \xrightarrow{\text{project}} \quad s_{\text{F#}} \\
    \text{where } s \text{ is a string}
\]

\( (\text{Embed-Bool}) \)
\[
    b_{\text{F#}} \quad \xrightarrow{\text{embed}} \quad b_{\text{JS}} \\
    \text{where } b \in \{\text{true, false}\}
\]

\( (\text{Proj-Bool}) \)
\[
    v_{\text{JS}} \quad \xrightarrow{\text{project}} \quad b_{\text{F#}} \\
    \text{where } b = \begin{cases} 
    \text{false} & \text{if } v \in \{\text{false, undefined, null, "", 0}\} \\
    \text{true} & \text{otherwise}
    \end{cases}
\]

\( (\text{Embed-Unit}) \)
\[
    () \quad \xrightarrow{\text{embed}} \quad \text{Undefined}
\]

\( (\text{Proj-Unit}) \)
\[
    \text{Undefined} \quad \xrightarrow{\text{project}} \quad ()
\]

\( (\text{Embed-Null}) \)
\[
    \text{null} \quad \xrightarrow{\text{embed}} \quad \text{Null}
\]

\( (\text{Proj-Null}) \)
\[
    \text{Null} \quad \xrightarrow{\text{project}} \quad \text{Unchecked.defaultof } \tau
\]
B.2 Collections

(Embed-Rec)

\[
\begin{align*}
\text{v}_{1,F} & \xrightarrow{\text{embed}} \text{v}_{1,JS} & \ldots & \text{v}_{k,F} & \xrightarrow{\text{embed}} \text{v}_{k,JS} \\
\{\text{lab}_1 = \text{v}_{1,F}; \ldots; \text{lab}_k = \text{v}_{k,F}\} & \xrightarrow{\text{embed}} \{\text{lab}_1 : \text{v}_{1,JS}, \ldots, \text{lab}_k : \text{v}_{k,JS}\}
\end{align*}
\]

(Project-Rec)

\[
\begin{align*}
\text{v}_{1,JS} & \xrightarrow{\text{project}_{\tau_1}} \text{v}_{1,F} & \ldots & \text{v}_{k,JS} & \xrightarrow{\text{project}_{\tau_k}} \text{v}_{k,F} \\
\{\text{lab}_1 : \text{v}_{1,JS}, \ldots, \text{lab}_k : \text{v}_{k,JS}\} & \xrightarrow{\text{project}} \{\text{lab}_1 = \text{v}_{1,F}; \ldots; \text{lab}_k = \text{v}_{k,F}\}
\end{align*}
\]

(Embed-List)

\[
\begin{align*}
\text{v}_{1,F} & \xrightarrow{\text{embed}} \text{v}_{1,JS} & \text{v}_{2,F} & \xrightarrow{\text{embed}} \text{v}_{2,JS} & \ldots & \text{v}_{k,F} & \xrightarrow{\text{embed}} \text{v}_{k,JS} \\
[\text{v}_{1,F}; \text{v}_{2,F}; \ldots; \text{v}_{k,F}] & \xrightarrow{\text{embed}} [\text{v}_{1,JS}, \text{v}_{2,JS}, \ldots, \text{v}_{k,JS}]
\end{align*}
\]

(Project-List)

\[
\begin{align*}
\text{v}_{1,JS} & \xrightarrow{\text{project}_{\tau}} \text{v}_{1,F} & \text{v}_{2,JS} & \xrightarrow{\text{project}_{\tau}} \text{v}_{2,F} & \ldots & \text{v}_{k,JS} & \xrightarrow{\text{project}_{\tau}} \text{v}_{k,F} \\
[\text{v}_{1,JS}, \text{v}_{2,JS}, \ldots, \text{v}_{k,JS}] & \xrightarrow{\text{project}_{\tau \text{ list}}} [\text{v}_{1,F}; \text{v}_{2,F}; \ldots; \text{v}_{k,F}]
\end{align*}
\]

(Embed-Array)

\[
\begin{align*}
\text{v}_{1,F} & \xrightarrow{\text{embed}} \text{v}_{1,JS} & \text{v}_{2,F} & \xrightarrow{\text{embed}} \text{v}_{2,JS} & \ldots & \text{v}_{k,F} & \xrightarrow{\text{embed}} \text{v}_{k,JS} \\
[\text{v}_{1,F}; \text{v}_{2,F}; \ldots; \text{v}_{k,F}] & \xrightarrow{\text{embed}} [\text{v}_{1,JS}, \text{v}_{2,JS}, \ldots, \text{v}_{k,JS}]
\end{align*}
\]

(Project-Array)

\[
\begin{align*}
\text{v}_{1,JS} & \xrightarrow{\text{project}_{\tau}} \text{v}_{1,F} & \text{v}_{2,JS} & \xrightarrow{\text{project}_{\tau}} \text{v}_{2,F} & \ldots & \text{v}_{k,JS} & \xrightarrow{\text{project}_{\tau}} \text{v}_{k,F} \\
[\text{v}_{1,JS}, \text{v}_{2,JS}, \ldots, \text{v}_{k,JS}] & \xrightarrow{\text{project}_{\tau \text{ array}}} [\text{v}_{1,F}; \text{v}_{2,F}; \ldots; \text{v}_{k,F}]
\end{align*}
\]

(Embed-Tuple)

\[
\begin{align*}
\text{v}_{1,F} & \xrightarrow{\text{embed}} \text{v}_{1,JS} & \text{v}_{2,F} & \xrightarrow{\text{embed}} \text{v}_{2,JS} & \ldots & \text{v}_{k,F} & \xrightarrow{\text{embed}} \text{v}_{k,JS} \\
(\text{v}_{1,F}, \text{v}_{2,F}, \ldots, \text{v}_{k,F}) & \xrightarrow{\text{embed}} [\text{v}_{1,JS}, \text{v}_{2,JS}, \ldots, \text{v}_{k,JS}]
\end{align*}
\]

(Project-Tuple)

\[
\begin{align*}
\text{v}_{1,JS} & \xrightarrow{\text{project}_{\tau_1}} \text{v}_{1,F} & \text{v}_{2,JS} & \xrightarrow{\text{project}_{\tau_2}} \text{v}_{2,F} & \ldots & \text{v}_{k,JS} & \xrightarrow{\text{project}_{\tau_k}} \text{v}_{k,F} \\
[\text{v}_{1,JS}, \text{v}_{2,JS}, \ldots, \text{v}_{k,JS}] & \xrightarrow{\text{project}_{\tau_1 \tau_2 \cdots \tau_k}} (\text{v}_{1,F}, \text{v}_{2,F}, \ldots, \text{v}_{k,F})
\end{align*}
\]
B.3 Exceptions

\[(\text{EMBED-Exc})\]
\[
\begin{align*}
&v_{1,F\#} \xrightarrow{\text{embed}} v_{1,\text{JS}} \\
&\cdots \\
&v_{k,F\#} \xrightarrow{\text{embed}} v_{k,\text{JS}} \\
\text{Exception}(v_{1,F\#}, \ldots, v_{k,F\#}) \xrightarrow{\text{embed}} \{\text{Message: "An F\# exception occurred"}, \\
&\text{Values: } [v_{1,\text{JS}}, \ldots, v_{k,\text{JS}}]\}
\end{align*}
\]

\[(\text{PROJ-Exc})\]
\[
\begin{align*}
&v_{\text{JS}} \xrightarrow{\text{project}} \text{JSExc}\text{eption}(v_{\text{JS}})
\end{align*}
\]

\[(\text{EMBED-raise})\]
\[
\begin{align*}
&E_{F\#} \xrightarrow{\text{embed}} E_{\text{JS}} \\
\text{raise } E_{F\#} \xrightarrow{\text{embed}} \text{throw } E_{\text{JS}}
\end{align*}
\]

\[(\text{PROJ-raise})\]
\[
\begin{align*}
&E_{\text{JS}} \xrightarrow{\text{project}} E_{F\#} \\
\text{throw } E_{\text{JS}} \xrightarrow{\text{project}} \text{raise } E_{F\#}
\end{align*}
\]

where \(\tau\) is the type \('a\) has been instantiated to in \text{raise: } \text{exn} \rightarrow \text{'a}.

B.4 Functions

\[(\text{EMBED-Func})\]
\[
\begin{align*}
&\text{arg}_{\text{JS}} \xrightarrow{\text{project}} \text{arg}_{F\#} \\
&(\text{fun } x : \tau \rightarrow e) \xrightarrow{\text{embed}} v \\
&\text{fun } x : \tau \rightarrow e \xrightarrow{\text{embed}} \text{function}(\text{arg}_{\text{JS}}) \{\text{return } v; \}
\end{align*}
\]

\[(\text{PROJ-Func})\]
\[
\begin{align*}
&E_{F\#} \xrightarrow{\text{embed}} \text{arg}_{\text{JS}} \\
&(\text{function}(x) \{e;\}) \xrightarrow{\text{project}} v \\
&\text{function}(x) \{e;\} \xrightarrow{\text{project}} \text{fun } E_{F\#} : \tau_1 \rightarrow v
\end{align*}
\]
Appendix C

Qualitative evaluation (additional details)

This appendix addresses success criterion 4 (convenient syntax). For this purpose, we use the cognitive dimensions (CDs) framework, as described by Blackwell and Green in [33, §5], in which each dimension describes an aspect of an information structure. It is typical to restrict the discussion of CDs to the most relevant ones (see [33, §5.5] for a sample case study by Blackweel and Green with 6 dimensions). Likewise, some dimensions are not addressed here because they are (a) the same for both systems (e.g., secondary notation), or (b) not an appropriate comparison due to differences in the languages used by the systems.

Here, we use the similar system Lua-ML and compare it with MiXture according to the CDs framework. We look at one short sample program for each system, each of which performs the same operation: embed a resolution theorem prover (from J. Harrison’s website [36], which includes code in F# and OCaml for his book [37]) into Lua and JavaScript, respectively.

C.1 Viscosity

Viscosity is the resistance to change of a system, hence viscous systems cause more work for the user whenever he tries to make a modification.

We can see that the explicit type information of Lua-ML (line 24 in OCaml in Listing [C.2] would require the user to make more changes should he decide to change the functions to embed or project. In these examples, we are only embedding one function, but this would become a tedious task if say all functions being embedded are changed to a version with side effects only (returning unit). MiXture requires no such type information (it obtains it through reflection), and hence we can conclude that Lua-ML is a more viscous system than MiXture. High-levels of viscosity break the train of thought and might cause low productivity in a developer when using two programming languages simultaneously.
C.2 Visibility / explicit dependencies

Visibility is the ability to view components easily.

The explicit initialization of the Lua interpreter in Listing C.2 makes the state and interpreter visible. On the other hand, MiXture does not require explicit initialization, although it does support similar functionality via the use of functions in the JSUtils module.

C.3 Diffuseness

Diffuseness is the verbosity of the system.

There is a trade-off between this dimension and visibility, and hence the diffuseness of MiXture is lower than that of Lua-ML.

C.4 Consistency

Consistency relates similar semantics with similar syntax.

Although not shown in the code listings, Lua-ML does not handle polymorphism in a
C.4. CONSISTENCY

consistent way. The type variables in polymorphic types must be instantiated with value (the equivalent to JSValue in MiXture, the type of Lua values in OCaml), which does not make use of OCaml’s parametric polymorphism. On the contrary, MiXture strives to assign the same semantics to polymorphic functions with the use of contracts. Furthermore, the type annotations for Lua-ML are non-native, while F# only needs native type-annotations in rare occasions. We can therefore conclude that MiXture is more consistent than Lua-ML.

```ocaml
#use resolution.ml
module T = Lua.Lib.Combine.T1 (Luaiolib.T)
module WT = Lua.Lib.WithType (T)

module C = Lua.Lib.Combine.C4 (* C4 == combine 4 code modules *)
    (Luaiolib.Make(T.TV1))
    (Luacamllib.Make(T.TV1))
    (WT (Luastrlib.M))
    (WT (Luamathlib.M))

module I = (* interpreter *)
    Lua.MakeInterp
    (Lua.Parser.MakeStandard)
    (Lua.MakeEval (T) (C))

module V = I.Value

let resolve (s:string) : bool list =
    resolution (parse s)

I.register_globals
    ["resolve", efunc
       (V.string (V.( **-> )) (V.list V.string) resolve)]
```

---

```lua
> resolve ("exists x. exists y. forall z."
   
   "(F(x,y) ==> (F(y,z) \ F(z,z))) \" 
   
   "((F(x,y) \ G(x,y)) ==> (G(x,z) \ G(z,z)))")

{[1]="t"}
```

Listing C.2: Embedding a resolution function into Lua and an interactive session transcript.
Appendix D

Sample code listings

These are some of the functions in MiXture.NEmbedding verbatim.

```csharp
// These functions are called by top level functions. They are not specialized for any type in particular.

/// <summary>Embeds a value into an <c>JSvalue</c>
/// <param name="x">The value that is being embedded</param>
/// <return>A value of type <c>_JSValue</c> which is the JavaScript equivalent of <c>x</c></return>
let rec embed (x:obj) : _JSValue =
    if box x = null then unit.embed()
    else embed_reflection x

/// <summary>Embeds a value into an <c>JSvalue</c>, using type information provided
/// <param name="ty">The <c>Type</c> value that specifies the type of the value being embedded</param>
/// <param name="x">The value that is being embedded</param>
/// <return>A value of type <c>_JSValue</c> which is the JavaScript equivalent of <c>x</c></return>
and embed_reflection (x:obj) =
    let ty = x.GetType()
    if ty = typeof<string> then string.embed(x:?>string)
    elif ty = typeof<float> then float.embed(x:?>float)
    elif ty = typeof<int> then int.embed(x:?>int)
    elif ty = typeof<bool> then boolean.embed(x:?>bool)
    elif ty.IsGenericType && ty.GetGenericTypeDefinition() = typedefof<list<_>> then
        embed_ienumerable(x :?> IEnumerable)
    elif ty.IsArray then embed_ienumerable(x :?> IEnumerable)
    elif (FSharpType.IsFunction ty) then
        embed_func(x)
    elif (FSharpType.IsTuple ty) then
        embed_tuple x
```
elif (FSharpType.IsExceptionRepresentation ty) then
  let fields = FSharpValue.GetExceptionFields x
  let js_fields = embed fields
  JSEngine.makeException(JSUtils.context, js_fields)

elif ty = typeof<System.Reflection.TargetInvocationException> then
  embed (x :?> System.Reflection.TargetInvocationException).InnerException

elif (FSharpType.IsRecord ty) then
  embed_record x
else
  let error_message = sprintf "Can't embed values of type %A" ty
  raise (EmbedException error_message)

and project_aux ty (x:obj) : obj =
  match x with
  | :? _JSValue as jx -> project_reflection ty jx
  | :? (_JSValue list -> _JSValue) as f ->
    if FSharpType.IsFunction (x.GetType()) && FSharpType.IsFunction ty then
      project_func ty f
    else
      raise (ProjectException "Function nonfunction")
  | _ -> raise (ProjectException "Trying to project a non JavaScript handle")

and project<'T> (x: _JSValue) : 'T =
  if JSEngine.isUndefined(x) then Unchecked.defaultof<_>
  else
    let ty = typeof<'T>
    project_aux ty x |> unbox<'T>

// These are auxiliary functions used by the above and are specific for each type
// to be embedded / projected.

and embed_func (x:obj) =
  // f is a function _JSValue -> _JSValue, which takes a JavaScript
  // Arguments object and returns the embedded result
let f (args: _JSValue) =
let arg = args |> JSUtils.get_all_JS_arguments JSUtils.context
let domain = FSharpType.GetFunctionElements (x.GetType()) |> fst
let projected_args =
  // if the function expects a tuple, process the array args
  // through project and build the appropriate tuple
  if FSharpType.IsTuple domain then
    let proj_args_array =
      Array.map2 project_reflection
        (FSharpType.GetTupleElements domain)
        arg
    [| FSharpValue.MakeTuple(proj_args_array, domain) |]
  else [| project_reflection domain (arg.[0]) |]
  try
    embed (Utils.call_object_function x projected_args domain)
  with
    | exn -> JSEngine.throwException(JSUtils.context, embed exn)
let callback = new JSEngine.FSharpFunction(f)
let gch = GCHandle.Alloc(callback)
let result = JSEngine.makeFunction(JSUtils.context, callback)
pinned_handles.[result] <- gch
result

and project_func ty (f:_JSValue list -> _JSValue) : obj =
let range = FSharpType.GetFunctionElements ty |> snd
if FSharpType.IsFunction range then
  FSharpValue.MakeFunction(ty,
    fun arg -> project_aux range (fun t -> f (embed arg :: t)))
else
  FSharpValue.MakeFunction(ty,
    fun arg -> project_aux range ( f [embed arg]))

and embed_record r =
let result = JSUtils.makeObjectLiteral(JSUtils.context)
let field_names, field_values =
  Utils.get_field_names (r.GetType()), Utils.get_field_values r
let field_writable = Utils.get_field_writable (r.GetType())
let field_values_embedded = Array.map embed field_values
let name_type_writable =
  Array.zip3 field_names field_values_embedded field_writable
  // set properties on result with field_names
  // as names and field_values as values
  Array.iter (JSUtils.setProperty result) name_type_writable
result

/// <summary>Projects a JavaScript value into an F# record</summary>
and project_record ty (x:_JSValue) =
let r_field_names = Utils.get_field_names ty
if JSEngine.isUndefined(x) then
    FSharpValue.MakeRecord(ty, Array.create r_field_names.Length null)
else
    let j_field_names: string[] =
        JSEngine.getOwnPropertyNames(JSUtils.context, x)
    |> project
    let length = r_field_names.Length
    if r_field_names.Length <> j_field_names.Length
    then
        let error_message =
            "The number of fields in the JavaScript object and \n            the F# record fields don't match"
        raise (ProjectException error_message)
    else
        // check that both names lists are a permutation of each other
        if (not (Utils.is_permutation r_field_names j_field_names)) then
            let error_message =
                "The names of the JavaScript object and \n                the F# record fields don't match"
            raise (ProjectException error_message)
        else
            let r_values = project_object_properties ty x r_field_names
            // this shouldn't fail because the values
            // have been projected as directed by ty
            try
                FSharpValue.MakeRecord(ty, r_values)
            with
                exn -> raise (ProjectException "Record creation failed")
Appendix E

Project proposal

Eduardo Munoz
Magdalene College
em487

Computer Science Tripos Part II Project Proposal

Multilanguage programming in F# and JavaScript

October 18, 2012

Project Originator: Eduardo Munoz

Project Supervisor: Tomas Petricek

Signature:

Director of Studies: Dr John Fawcett

Signature:

Overseers: Dr S. Teufel and Dr J. Crowcroft

Signatures:
Introduction and description of the work

Selecting the programming language to use when implementing a certain algorithm is crucial to the success of the project. For this reason, large scale software systems tend to be written in several languages. However, the interactions between these components may be an issue. There are several approaches to tackle this:

1. **Foreign function interfaces (FFI)**. Mechanism by which a program written in one language may call procedures from a program in a different language. The Java Native Interface is an example of a FFI.

2. **Multilanguage runtimes**. Several programming languages target the same architecture, allowing a richer interaction between the languages: e.g., inherit classes in one language defined in another, call higher-order functions, etc. Examples of multilanguage runtimes targeting the Java Virtual Machine are Java, Scala, Jython and JRuby; all programs written for the .NET framework target the Common Language Runtime.

3. **Embedded interpreters**. Implement an interpreter of the target language in the host language and use type-indexed embedding and projection algorithms \[3\]. Lua-ML is an example of this technique, where a Lua interpreter is implemented in OCaml \[4\].

In this project, we intend to explore the ideas described by Matthews and Findler \[8\], for an ML-like language and JavaScript\[1\]. This includes producing an implementation of the lump embedding and the natural embedding. The former concept is related to FFI frameworks, where each environment sees foreign values across the boundary as “lumps” (values with an opaque type). The natural embedding provides a richer interoperability between the two languages, allowing values to be converted across the boundary from one language to the other. The implementation for the natural embedding can be seen as FFI with some properties of multilanguage runtimes.

An example of using the lump embedding (note that the syntax and types haven’t been decided yet, this is for illustrative purposes):

(* eval_js x : Lump list -> Lump is provided by the framework *)
let apply_lump f x = eval_js([f,x]);
let succ:Lump = JS("function(a) {return a+1;}")
in apply_lump(succ, JS("3"));

---

\[1\]In this project, JavaScript is not utilized as client-side language in the browser, but as a general purpose language.
In this case, the ML-like language interacts JavaScript values (of type Lump), by applying the value 3_{JS} to the JavaScript function succ.

An example of using the natural embedding (note that the syntax and types haven’t been decided yet, this is for illustrative purposes):

```ml
let test (f:string->unit) :unit = f("testing")
in test(JS(string, unit, "function(s) {print(s)}"));
```

We can see the natural embedding allows a richer interaction between the two languages. In the example above, a test function is defined in ML, which takes a function f and passes it the string "testing". An anonymous JavaScript function is passed as the f parameter to test.

Note that JS acts both as a language boundary in both types of embedding and as a value constructor for JavaScript values.

## Starting point

This project will involve material from *Types* (deal with language boundaries), *Semantics of Programming Languages* (specify the behavior of the framework), *Compiler Construction* (analyze the runtime of the JavaScript engine to integrate it) and *Foundations of Computer Science* (functional programming, ML).

Apart from (re-)familiarizing myself with that material, I will also study research papers (mainly the ones cited in this proposal).

## Resources required

In this project, I will make use of a JavaScript engine. My initial intention is to use Google’s V8 engine, but I will leave open the possibility of using a different engine in case V8 proves to be more problematic than expected. The implementation will be completed on my own laptop using an ML-like functional programming language (F#).

It is possible that a contracts library will be required in JavaScript.
Substance and structure of the project

The goal of this project is to design and implement a framework for the multilanguage programming paradigm. Work will be done in the following areas:

Type system

JavaScript is an untyped\(^2\) language, while F# has strong static typing with type inference.

To be done

1. Define types for the lump and the natural embedding.
2. Handle types at the boundaries of the programming languages, making sure the type soundness of F# is preserved (possible use of \textit{contracts}).

Evaluation rules

Evaluation rules differ in both languages. For instance, JavaScript allows the evaluation of two empty lists \[\text{[14]}\]; F# however forbids this operation because it doesn’t type check:

\begin{verbatim}
(JavaScript)
j> [] + []

// (empty string)

---------------------
(F#)
> [] + [];

[] + [];;
-----^`

stdin(1,6): error FS0001: None of the types ’a list, ’a list’ support the operator ’+’
\end{verbatim}

\(^2\)There is some confusion with the terms \textit{dynamically typed} and \textit{untyped}. In the academic literature, the term dynamically typed was introduced much later than untyped to mean the same concept. Perhaps the best categorization for JavaScript is \textit{weakly dynamically typed}.\)
To be done

1. Define new rules for the evaluation of programs where language boundaries exist (e.g., F#'s runtime calls a JavaScript function with an ML-native value).

2. Define the behavior of the framework if mismatching types are used across the boundaries.

Values

This is important for the natural embedding (conversion of primitive datatypes to allow foreign values to have a native type). For instance, JavaScript treats all numbers as floating point numbers:

(JavaScript)
```javascript
js> 1000000000000000000 + 1
1000000000000000000
js> 4611686018427387903 + 1
4611686018427388000
```

(OCaml)
```ocaml
# 1000000000000000000 + 1;;
- : int = 1000000000000000001
# 4611686018427387903 + 1;;
- : int = -4611686018427387904
```

To be done

1. Define conversion strategies for values of primitive types.

2. Provide glue code to cope with cross-boundary calls.

Correctness

Design automated tests of correctness for the lump and natural embeddings.
Possible extensions

1. Syntax-check JavaScript using F# type providers.
2. Allow passing of native objects between the languages. JavaScript objects could be represented as records in F#.
3. Allow passing other native non-primitive values: collections, exceptions, etc.

Evaluation strategy

Quantitative:

- Compare the performance difference between the system implementing the lump embedding and the natural embedding.
- Estimate the relationship between the number of “foreign” boundary crossings and the execution time.
- Compare the performance difference with other systems that allow some interoperability between F# and JavaScript.

Qualitative:

- Show the expressiveness of the system.
- Evaluate of correctness, by executing some tests.

Backup strategy

All source files (code and \LaTeX) are in my local machine in a Git repository, which is hosted on Bitbucket and also replicated to the Desktop Services (DS) servers provided by the Computer Laboratory\footnote{with hostname linux.cl.ds.cam.ac.uk}. The Git repository will be useful when writing the dissertation as it will be used as a work log.
Success criteria

1. The lump embedding implementation should be able to pass values from JavaScript to F# and then pass them back.

2. The resulting framework should not take significantly more time than executing the respective monolingual runtimes.

3. The natural embedding should be able to pass a function from JavaScript to F# (and vice versa) and invoke it on the other side of the boundary. It should also be able to convert primitive datatypes between the two languages.

4. A convenient syntax for multilanguage programming has been designed.

5. Tests of correctness have been passed.

Timetable and milestones

Michaelmas


Install F# on my laptop (using the Mono framework for .NET); install Windows as a fallback. Revise ML and become familiar with the differences in F#. Keep reading research papers and articles about the subject. Research JavaScript engines and decide which one offers the best API to access JavaScript values.

*Milestone:* Have a working development environment and a chosen JavaScript engine.

Preparation II: 01.11.2012 - 14.11.2012

Investigate and test accessing values from the JavaScript engine into F#. Decide on a specific syntax for embedding JavaScript inside F#. If time allows, start the implementation of the lump embedding.

*Milestone:* The plan for the implementation is finished.

Implement JavaScript values as Lumps in F#. Finish the lump embedding implementation. **Milestone:** Have a working version of the lump embedding. Start implementing the natural embedding.

Christmas break


Implement conversion of primitive datatypes, using the JavaScript engine chosen in Preparation I. **Milestone:** F# can interact with primitive datatypes from JavaScript and vice versa.


Implement wrappers that allow treating JavaScript functions as F# functions. Start designing correctness tests and begin implementing them. **Milestone:** F# can now call JavaScript functions.


Investigate the use of contracts to implement wrappers that allow treating F# functions as JavaScript functions. Test current implementation with the correctness tests; modify the implementation until all tests are passed. **Milestone:** JavaScript can now call F# functions and the correctness tests are passed.

Lent

Extensions: 10.01.2013 - 23.01.2013

Implement some extensions, depending on the progress of the core. The core must be in a state in which the implementation section can be written and the evaluation can be performed. If this is not the case, finalize any parts of the implementation which proved to be harder than expected. **Milestone:** the core implementation of the project is finished.
Progress report & write-up I: 24.01.2013 - 06.02.2013

Write the progress report and prepare the progress presentation slides. Write the introduction, preparation and implementation sections.
*Milestone:* the dissertation document has been started.

Write-up II: 07.02.2013 - 20.02.2013

Finish off the following sections: preparation and implementation. Take evaluation metrics.
*Milestone:* preparation and implementation are finished. Evaluation data is gathered.

Write-up III: 21.02.2013 - 06.03.2013

Write the evaluation and conclusion sections of the dissertation.
*Milestone:* a draft of the whole dissertation is now complete.

Write-up IV: 07.03.2013 - 20.03.2013

Make formatting changes to the document. Send final draft to my supervisor and make small adjustments if suggested.
*Milestone:* the document’s formatting is final.

Easter break

Final editing and submission: 21.03.2013 - 03.04.2013

Make any last-minute minor changes to the dissertation / code if suggested by my supervisor or DoS.
*Milestone:* have the dissertation approved, hand it in.