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:

1 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 [1]. LuaML is an example of this technique, where a Lua interpreter is implemented in OCaml [2].

In this project, we intend to explore the ideas described by Matthews and Findler [3], 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));

> val it: JS(4) : Lump
```

[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):

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

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.

2 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).

3 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.

4 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:

4.1 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 *contracts*).

4.2 Evaluation rules

Evaluation rules differ in both languages. For instance, JavaScript allows the evaluation of two empty lists [4]; F# however forbids this operation because it doesn't type check:

```
(JavaScript)
js> [] + []

// (empty string)

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

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

stdin(1,6): error FS0001: None of the types ''a list, 'a list' support
                          the operator '+'
```

[2]There is some confusion with the terms *dynamically typed* and *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 *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.

4.3 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)
js> 1000000000000000000 + 1
1000000000000000000
js> 4611686018427387903 + 1
4611686018427388000
--------------------
(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.

4.4 Correctness

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

5 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.

6 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.

7 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[3]. The Git repository will be useful when writing the dissertation as it will be used as a work log.

[3]with hostname `linux.cl.ds.cam.ac.uk`.

8 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.

9 Timetable and milestones

Michaelmas

Preparation I: 18.10.2012 - 31.10.2012

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: 01.10.2012 - 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.

Implementation I: 15.11.2012 - 28.11.2012

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

Implementation II: 29.11.2012 - 12.12.2012

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.

Implementation III: 13.12.2012 - 26.12.2012

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.

Implementation IV: 27.12.2012 - 09.01.2012

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: 10.01.2012 - 23.01.2012

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: 24.01.2012 - 06.02.2012

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: 07.02.2012 - 20.02.2012

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

Write-up III: 21.02.2012 - 06.03.2012

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

Write-up IV: 07.03.2012 - 20.03.2012

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: 21.03.2012 - 03.04.2012

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.

References

[1] N. Benton, "Embedded interpreters," *J. Funct. Program*, vol. 15, no. 4, pp. 503–542, 2005. [Online]. Available: http://dx.doi.org/10.1017/S0956796804005398

[2] N. Ramsey, "ML module mania: A type-safe, separately compiled, extensible interpreter," *Electr. Notes Theor. Comput. Sci*, vol. 148, no. 2, pp. 181–209, 2006. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2005.11.045

[3] J. Matthews and R. B. Findler, "Operational semantics for multi-language programs," *ACM Trans. Program. Lang. Syst*, vol. 31, no. 3, 2009. [Online]. Available: http://doi.acm.org/10.1145/1498926.1498930

[4] ECMA, *ECMA-262: ECMAScript Language Specification*, 3rd ed. pub-ECMA:adr: ECMA (European Association for Standardizing Information and Communication Systems), Dec. 1999. [Online]. Available: http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM