F# Web Tools: Rich client/server web applications in F#

(Unpublished draft)

Tomáš Petříček
Charles University in Prague
Faculty of Mathematics and Physics
tomas@tomasp.net

Don Syme
Microsoft Research,
Cambridge
dsyme@microsoft.com

Abstract

"Ajax" programming is becoming a de-facto standard for certain types of web applications, but unfortunately developing this kind of application is a difficult task. Developers have to deal with problems like a language impedance mismatch, limited execution runtime for the code running in web browser on the client-side and no integration between client and server-side parts that are developed as a two independent applications, but typically form a single and homogenous application. In this paper we present the first project that deals with all three mentioned problems but which still integrates with existing web development technologies such as ASP.NET on the server and JavaScript on the client. We use the F# language for writing both client and server-side part of the web application, which lets us develop client-side code in a type-safe programming language using a subset of the F# library, and we provide a way to write both server-side and client-side code as a part of single homogeneous module defining the web page logic. This code is executed heterogeneously, part as JavaScript on the client, and part as native code on the server. Finally we use monadic syntax for the separation of client and server-side code, tracking this separation through the F# type system.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming

General Terms Languages

Keywords ML, F#, Meta-Programming, Web Development, Ajax

1. Introduction

This paper is concerned with the following question: how do we bring the goodness of ML to web programming? That is, how do we bring the benefits of typed functional programming to the tasks that occupy the lives of the thousands of programmers who currently write web applications in JavaScript (on the client side) and PHP/VisualBasic/C#/Java/Ruby (on the server side). This class of applications can, broadly speaking, be called Ajax[1] applications. We show that the F# dialect of ML can be used to build homogeneous (i.e. single-language) type-checked client/server Ajax applications that are executed heterogeneously, as JavaScript in the browser and as native code on the server. Among other things this allows the construction of client-side functionality that takes advantage of the particular symbolic-programming strengths of ML.

As first summarized in [1] and later clarified in [2] Ajax applications typically perform the following tasks: they present data using (X)HTML and CSS in a web browser, they use asynchronous requests back to the server to dynamically load data on demand, and they update the displayed data using JavaScript code running in the browser. One important observation made in [2] is that this definition discusses only the client-side part of the application (running in a web-browser), but ignores the code running on the server-side, while in most of the situations these two parts are actually inseparable parts of a single application. Indeed, while an Ajax application executes primarily on the client side, it is authored on the server and served to the client in response to an initial HTTP request. Hence a program written in one language (the server-side program) must serve and interoperate with a program writer in another (the JavaScript client-side program). This somewhat confused mix of multiple languages, staged computation and program generation means there is a real need for tools, languages and frameworks that make the development of these applications easier.

In this paper we show how to solve three of the fundamental difficulties of web programming: the heterogeneous nature of execution, the discontinuity between client and server parts of execution and the lack of type-checked execution on the client side. We use the F# language [24], making use of recent additions including meta-programming via "quotations" [10], simpler manipulations of quotations via active patterns [11], the combination of object-oriented programming with functional programming [23, 24], and a monadic syntax akin to that of Haskell [24]. On the server side we use ASP.NET framework [9], because it is easily accessible from F#, though we believe that our approach could be used with many other web development tools, accessible via Mono [25]. We also discuss what language features are crucial for supporting particular features of the presented work.

From a language point of view, our work presents the following interesting aspects:

- We use meta-programming to write code that is executed across heterogeneous environments using a single homogenous language. As far as we are aware, we are first to use it for a web development scenario.
- We present the first translator from an ML-family functional language to JavaScript, including the translation of ML core language constructs and as well as a subset of the F# and .NET libraries. We also show how

[1] Ajax stands for Asynchronous JavaScript and XML, The Ajax name first appeared in [1].

native JavaScript components[3] can be accessed from the source language in a type-safe way.

- We discuss approaches that we are investigating for verifying that the part of the code which is intended to run on the client-side (and so will be translated to JavaScript) uses only functions and types that have a corresponding JavaScript implementation.

The use of F# for developing both client-side and server-side parts of the application together with the fact that both parts can be written in single file (and in the case of class-based components even in a single component) allows us to achieve many interesting things from a web-development point of view as well. The contributions of this work from the web-development perspective are:

- We allow the use of F# "asynchronous workflows" (essentially monadic syntax for the continuation monad) in client side code, which makes it very easy to write non-blocking code (for example repeated server-side polling) that would be otherwise be written explicitly using events.
- We present a mechanism for managing client-side state changes performed by server-side components during asynchronous callbacks to the server.
- We show possible ways for developing a component based web development framework where interactions between components can be defined for both server and client side code in a uniform way.

In the rest of the paper we first discuss the difficulties of the Ajax-style web development (§1.1), provide an overview of relevant F# language constructs and further discuss the background of web development in general (§0). Next we present an example where we use our project to write an application running in a web browser performing parsing and symbolic manipulations of the entered mathematical expressions (§2) and follow the example with detailed discussion of the used techniques (§3). In §5 we present an example of a data-driven web application for organizing lectures and describe aspects of our work important for that kind of applications. Finally, in §5 we discuss related work, mention possible future work and discuss the repeatability of the presented solution by reviewing all the important contributions of our work and looking at the language features that are important to make particular parts possible.

1.1 Why Web Applications are Hard

We start with a discussion of the reasons why developing Ajax-style web applications is difficult.

The first reason is that the environment available on the client-side is limited in many ways – for better or for worse, the only language available in web browsers is JavaScript, which is not suitable for solving many of the problems that arise when developing complex web applications. There are also many incompatibilities between different JavaScript language and client-side library implementations. For these reasons some people (e.g., Meijer [4]) view JavaScript and related libraries only as an assembly language and runtime for compiling more complex languages and applications. This is also a view adopted in our work. Put bluntly, we take as axiomatic the need to execute some ML code in JavaScript some of the time. This is in the same way we assume the need to execute some code as x86 machine instructions. This far-reaching observation changes the whole way we should think about designing compilation hierarchies and tool support for our languages. Fortunately, the fundamental need to embrace heterogeneous execution has already been tackled in the context of F# [10].[4]

The second problem is the discontinuity between server and client-side parts of the application – even for very simple applications both parts have to be written separately, in different languages and with explicit way of communicating between both sides. There are several projects that deal with language impedance mismatch [4, 5, 6], but only a few projects go beyond this and deal also with the separation of client and server-side code (Links language [3]).

Another problem appears when using server-side frameworks such as ASP.NET or PHP. In these frameworks components (e.g. a calendar control, or a data list) exist only in the server-side code and interactions between these are executed only when processing whole page requests from the client. On the client-side, the components behave like a black-box and there is no way to specify client-side interactions that are an important aspect of any realistic interactive web application. For example, a calendar control may need to interact with a data list control to show information according to the selected date. If the information is already available on the client then this interaction should not involve "going back to the server" to rebuild the entire page, as is required by ASP.NET, and achieving this kind of "smoothness" is part why people resort to Ajax-style programming in JavaScript. Some component-based frameworks make it possible to define client-side components as well (for example [7]), but even with these extensions it is still impossible to define both server and client-side properties of a component using a single interface.

Background

In this section we first briefly discuss the F# language [8], its support for meta-programming including differences from a version described in [10], along with F# active patterns [11], used when manipulating F# meta-programs. We also informally describe a recent addition to F#: a monadic syntax akin to that of Haskell and which is very important for our work. We also introduce web development in general and changes in the understanding of web caused by a rise of Ajax based interactive web applications.

1.2 F# Language and Runtime

1.2.1 Core Language

The F# language is documented on the F# web site [8] and [10] presented following description: "F# is a multi-paradigm .NET language explicitly designed to be an ML suited to the .NET environment. It is rooted in the Core ML design and in particular has a core language largely compatible with OCaml." For our work it is important that F# supports mixed functional/OO programming and that it interoperates with the .NET platform, which allows us to use some parts of ASP.NET framework in our work.

Object oriented constructs in F# are compatible with OO support in the .NET CLR, which means that F# supports single implementation inheritance, multiple interface inheritance, subtyping, and F# object types can have fields, constructors, methods and properties (a property is just a syntactic sugar for getter and setter methods). An example that introduces the F# syntax for object types is shown in Figure 1.

[3] By native JavaScript components we mean JavaScript libraries that don't have corresponding representation in the source library (in our case F# and .NET class libraries)

[4] At some point another execution technology may replace JavaScript as the ubiquitous, "baseline" execution machinery on the client side of web applications. However, even if that happens it is likely that the overall client/server application will still be heterogeneous, hence the lesson still remains.

```
type MyCell(n:int) =
  let data = n + 1
  member x.Data = x.data
  member x.Print() = printf “%d” x.data
  static member FromInt(n) = new MyCell(n)
```

Figure 1. Example of an object type with a field called `data`, a property called `Data`, an instance method `Print`, a static method `FromInt` and an implicit constructor that initializes the value of the field.

1.2.2 F# Intensional Meta-Programming

The meta-programming capabilities of F# and .NET runtime can be viewed as a two separate and orthogonal parts. The .NET runtime provides a way for discovering intensional (i.e. abstract syntax trees) for all the types and top-level method definitions in a running program: this API is called `System.Reflection` and is akin to reflection in Java. F# “quotations” [10] provide a way for working with selected F# expressions in a similar way.

An important part of the .NET reflection mechanism is the use of *custom attributes*, which can be used to annotate any program construct accessible via reflection with additional metadata. The following example demonstrates the syntax for attributes in F# by adding `Documentation` attribute to a top-level definition:

```
[<Documentation("Adds one to a given number")>]
let addOne x = x + 1
```

F# quotations form the second part of the meta-programming mechanism, by allowing the capture of type-checked F# expressions as structured terms. There are two ways for capturing quotations; the first way is to use quotation literals as demonstrated in the following example (adopted from [10]):

```
<@ 1 + 1 @>              : int expr
<@ (fun x -> x + 1) @>   : (int -> int) expr
```

The second option (which is used more often in our work) is by explicitly marking top-level definitions with an attribute that instructs the compiler to capture the quotation of the entire definition body:

```
[<ReflectedDefinition>]
let addOne x =
  x + 1
```

The quotation of a definition (which can be either function or class member) annotated using the `ReflectedDefinition` attribute is then made available through the F# quotation library at runtime using the reflection mechanism described earlier, however the definition is still available as compiled code and can be executed.

The F# quotations also provide mechanism for splicing values into the quotation tree, which is useful mechanism for providing input data for further quotation processing. The operator for splicing values is the Unicode symbol (§) as demonstrated in the following example, where we use it for embedding a value that represents a database table[6]:

```
<@ §db.Customers
     |> filter (fun x -> x.City="London") @>
```

Programmatic access to F# quotation trees uses F# active patterns [11], which allow the internal representation of quotation trees to be hidden while still allowing the use of pattern matching as a convenient way to decompose and analyze F# quotations terms. One of the examples presented in [10] written using active patterns looks as follows:

```
let sinExpr = <@@ sin @@>;;

// Match with top-level definition pattern:
let sinTopDefData, sinTypeArgs =
  match sinExpr with
    | AnyTopDefnUse(res) -> res
    | _ -> failwith "no match";;
```

1.2.3 F# Monadic Syntax

The last feature of F# that is important for our work is the F# monadic syntax, which was introduced recently and hasn’t been described in the literature before. Since it is not part of our work, we will not describe it fully, but we need to introduce it at informally, because we use it later in this paper.

Properties of a monadic type are defined by a builder type which specifies type of a monadic value and behavior of the *bind* and the *return* operators. The following code shows signature of an example type `MBuilder`, which builds a monad of type M:

```
// Signature of the builder for monad M
type MBuilder with
 member Bind   : M<'a> * ('a -> M<'b>) -> M<'b>
 member Result : 'a -> M<'a>
 member Let    : 'a * ('a -> M<'b>) -> M<'b>
```

The `Bind` and `Result` members specifies standard monadic operators, the `Let` operation is used when binding value of ordinary type in a monad (in most situations it could be expressed using `Bind` and `Result`, but F# defines it as a separate operation to give more control over binding). A sample implementation of a builder for a trivial monadic type is shown below:

```
type M<'a> = unit -> 'a
type MBuilder() =
  member m.Bind(a,b) : M<'b> = b(a())
  member m.Result(a) : M<'a> = (fun () -> a)
  member m.Let(v,f)  : M<'b> = f v

// Instance of the builder will be used later
let mb = new MBuilder()
```

Having a monadic builder, we can now use a syntactic extension that makes it possible to write code that uses the monadic operations in a similar way as you would write ordinary F# code, as demonstrated in following example:

```
mb { let  ordinary = 5
     let! bound = mFunc()
     ->   ordinary + bound }
```

Here, `mb` is an instance of type `MBuilder`[7] and `mFunc` is a function with signature `unit -> M<int>`. When using a ordinary value the `let` or `do` keywords are used (and these will be translated to a call to the `Let` operator), to bind a monadic value, we can use `let!` and `do!` keywords. The code de-sugars to explicit calls to monadic binders:

```
mb.Let(5, fun ordinary ->
  mb.Bind(mFunc(), fun bound ->
    mb.Result(ordinary + bound)))
```

In this paper we make use of the following constructs from the monadic syntax:

```
expr =
  | ident { mexpr }
```

[6] This is a simplified example demonstrating a mechanism used in the latest version of the F# database access library (FLinq) presented in [10].

[7] The main reason why the instance needs to be specified is that it gives the F# compiler knowledge about the monadic builder that should be used in the monadic expression.

```
mexpr =
   | let! pat = expr in mexpr   monadic bind
   | do! expr                   monadic "unit" bind
   | let pat = expr in mexpr    regular bind
   | do expr                    regular "unit" bind
   | if expr then mexpr else mexpr      conditional
   | match expr with (pat -> mexpr)+    match
```

F# monads differ from Haskell monads partly in that it is not possible to write code that is generic over the kind of monad being used. We haven't found that a problem in practice, mainly because monads are used less frequently in F# programming than in Haskell.

1.3 Web Programming

In the recent years, the use and understanding of the World Wide Web has changed and applications that work on principles different to those originally envisaged for the web are appearing [13]. These applications are based on immediate interaction with the user and combine the behavior of desktop applications and traditional web applications.

Developing highly interactive applications[8] is a difficult task, partly because this requirement wasn't envisaged in the original web standards, e.g., many problems are caused by the stateless nature of the HTTP protocol. The use of the page as a unit of display also results in awkward techniques where reloads of an entire application in the web browser are required to updated simple elements of the client-side display.

Many of the frameworks that are currently used for web development try to abstract from the underlying technologies. A very common abstraction is composing pages from reusable server-side components that hide the complexity of building complex user interfaces. The application then defines only interactions between the components. In the presented work we use some parts of ASP.NET, which follows this model, however we use only the component model and only in one part of our work, so most of the work is independent to this technology and the rest could be easily adapted to use different model.

1.3.1 Control-flow in Web Applications

Ajax was introduced [1] as a name for a set of technologies that allow writing interactive web applications that share some common aspects with desktop applications. Most of the technologies that form Ajax were available before the name appeared and now the name is used more to describe the control flow of a class applications than the particular technology that is used to implement that control flow.

The differences between control flows are demonstrated at Diagram 1, which represents initial request and one page update traditional web application and Diagram 2, which represents the same operation in an Ajax application.

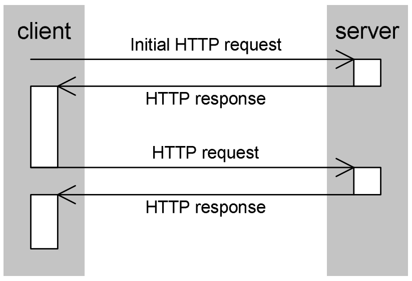


Diagram 1. Control flow of traditional web application

[8] Here we understand application only as one of the possible types of web contents, which have much closer to a desktop applications than to a page that presents (to some point) static data and we will follow this understanding in the rest of the paper.

In the traditional web application (Diagram 1) the client first requests a page (using HTTP request), the server code is executed and response, which consists only of data (HTML markup, images, etc.) is sent to the client. When the client wants to view other data or wants any update from server, it needs to send another HTTP request and refreshes the entire page.

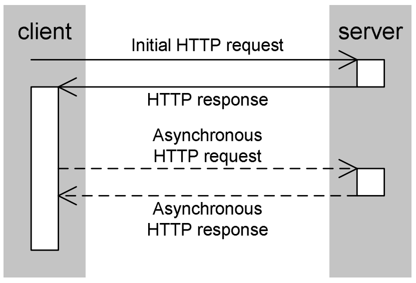


Diagram 2. Control flow of interactive "Ajax" application

In an Ajax application (Diagram 2), the client initiates with a request to the server, but the answer from server consists of data and client-side JavaScript code, which will start executing on the client side as soon as it is received. When the user wants to display different data or perform any interaction with server, the client-side code sends an asynchronous request to the server, which generates a response and sends it back to the client. The client than processes the response (using JavaScript) and updates the displayed content according to the data it received.

1.3.2 Client-Side Runtime Environment

The previous section shows that in Ajax based applications, the client-side part of the application is getting more important and more complex as well, because it is responsible for sending asynchronous requests to the server and processing the responses, updating the user interface on the client and also performing possible computations to minimize server workload.

A typical requirement for a web application is to work correctly in all frequently used browsers and platforms, which limits the choice to JavaScript[9]. Some projects (for example [15]) attempt to provide extended environment running on multiple platforms, but the range of platforms supporting JavaScript is still much wider.

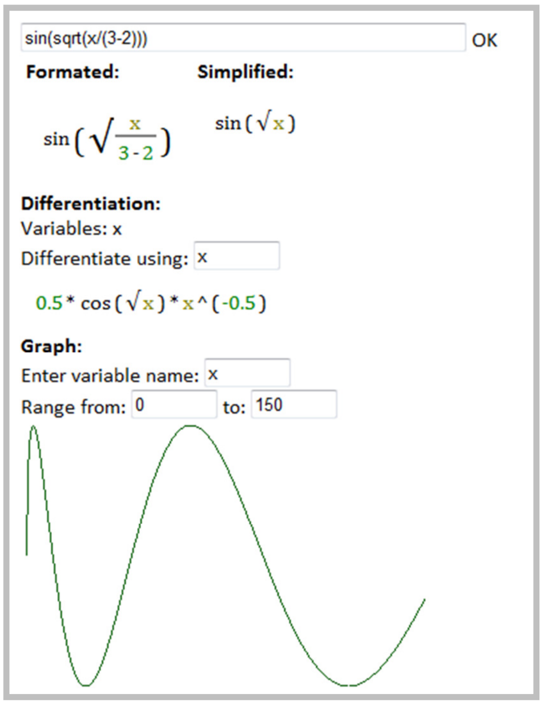
2. Example: Web Symbolic Manipulation

In the first example we use F# to develop an application running as a JavaScript code in a web browser that performs tasks that are traditionally easy to solve in a functional language. The presented application performs tokenization and parsing of the entered text and produces an AST representing elementary mathematical expressions. Further, the application performs symbolic differentiation and simplification of the expression, everything running "live" in the web browser, despite the fact that the application is originally authored as a server-side program. The running application as well as source code for all the samples mentioned in this paper is available at our web site (http://tomasp.net/fswebtools [16]). Figure 2 presents a screen-shot of a running application.

[9] By JavaScript we mean implementation corresponding to the ECMA-262, edition 3 [14] standard published in 1999, which is supported in most of the web browsers. The draft for JavaScript 2.0 (edition 4 of the ECMA standard) was recently published, but it is unlikely that it will be supported by all main-stream web browsers soon.

2.1 Parser Functions

The parser is implemented as a set of functions in a single F# module. The signatures of exported functions as well as a type used to represent AST tree are shown in Figure 3.



sin(sqrt(x/(3-2))) OK

Formated: $\sin\left(\sqrt{\frac{x}{3-2}}\right)$

Simplified: $\sin(\sqrt{x})$

Differentiation:

Variables: x

Differentiate using: x

$0.5 * \cos(\sqrt{x}) * x \hat{} (-0.5)$

Graph:

Enter variable name: x

Range from: 0 to: 150

Figure 2. Symbolic manipulation code written in F# and running as JavaScript code in the web browser

```
type AstNode =
  | Number   of float
  | Var      of string
  | Binary   of char * AstNode * AstNode
  | Unary    of char * AstNode
  | Function of string * (AstNode list)

[<NeutralSide>]
module Parsing =
  val tokenize : string -> Token list
  val simplify : AstNode -> AstNode
  val prettyPrint : AstNode -> string
  val parse : Token list -> AstNode
  val getVars : AstNode -> ResizeArray<string>
  val eval : AstNode * (string -> float) -> float
  val differentiate : AstNode * string -> AstNode
```

Figure 3. Signatures of functions available in the sample and types used for representing AST.

The module is marked using a `NeutralSide` attribute, which means that functions contained in it are implemented using library functions and types that are available on both the client-side and the server-side execution environments – it is not using any types or functions that could be executed only at client-side (like displaying browser dialog box) and no server-side only code (for example accessing the database). The verification of these rules is a separate topic discussed in §3.2.

The following example shows a simple evaluation function written in F# and we later show the JavaScript code generated by our translator:

```
let evaluate(nd, varfunc:string -> float) =
  let rec eval = function
    | Number(n) -> n
    | Var(v) -> varfunc v
    | Binary('+', a, b) ->
        let (ea, eb) = (eval a, eval b)
        ea + eb
    | _ -> failwith "unknown"
  eval nd
```

The code has a few interesting aspects from compiler point of view – it uses higher order functions to read values of variables in the expression, it defines inner recursive functions and it is written using pattern matching on algebraic data type representing the AST. Our translator produces following code[11]:

```
function evaluate(nd, varfunc) {
  var eval = (function (matchval) {
    if (true==matchval.IsTag('Number'))
      return matchval.Get('Number', 0);
    else {
      if (true==matchval.IsTag('Var'))
        return matchval.Get('Var', 0);
      else {
        if (true==(matchval.IsTag('Binary') &&
             createDelegate(this, function() {
                 var t = matchval.Get('Binary', 0);
                 return t = '+'
               })())) {
          var c = matchval.Get('Binary', 0);
          var a = matchval.Get('Binary', 1);
          var b = matchval.Get('Binary', 2);
          var t = CreateObject(new Tuple(), [a,b]);
          var ea = t.Get(0);
          var eb = t.Get(1);
          return ea + eb
        } else {
          return Lib.Utils.FailWith("unknown op.");
        }
      }
    }
  })
  return eval(nd);
}
```

Figure 4. JavaScript code generated for the F# evaluate function.

Since JavaScript supports first-class functions directly, there is no special handling required in the translator. F# data types like tuples and algebraic data types are not directly supported in JavaScript, so the translator uses two simple objects to emulate them. The type representing all algebraic data types supports method IsTag, which tests if algebraic data type contains specific value and Get, which reads one of the stored values. The type representing tuple has only Get method for accessing specified item (this implementation relies on the fact that JavaScript has runtime type checking).

The technique used for translating F# to JavaScript is further described in section §3.1, where we mention other problematic topics, like difference between expression and statement in JavaScript, different variable scoping etc.

2.2 Integrating Client and Server Code

One of the goals of our work is to make it possible to write a single file of code that overall represents the behavior of one web application, but where portions run in multiple different environments. In the parser example, the page uses functions from the Parsing module described above and contains following functionality:

- The entered expression is visualized using HTML (the visualization is being updated as you enter the expression).
- The expression is simplified and symbolic differentiation is calculated as the expression is entered.

[11] In our current implementation, the generated code is more complex, due to the use of the `if` as an expression in functional languages (all conditional expressions in this example were changed to statements with imperative return); however we plan to implement this simplification as a special case.

- When expression changes, it is sent to the server-side, which renders a graph of the function and sends response with the generated image URL.

First two operations involve executing only client-side code (a callback utilizes functions from the `Parsing` type declared earlier), but for the third operation, the client-side code needs to collaborate with the server side-code (a server-side function draws a graph and sends its address back to the client). The following code shows initial portions of three F# functions: `TextChanged` and `Process` are executed on the client side and a function called `GenerateImg` that draws a graph of the entered function, executed on the server-side. The functions are all members of the same "page" object. Calls between the client and server-side will be discussed shortly.

```
member this.TextChanged (s:obj, e:EventArgs) =
 client
  { let t = Parsing.Tokenize(this.txtInp.Text);
    do! this.Process(Parsing.Parse(tokens)) }

member this.Process (ast:AstNode) =
 client
  { ... }

member this.GenerateImg (expr:string) =
 server
  { ... }
```

Figure 5. Example shows subset of page interaction logic of the symbolic manipulation application. It contains one server and two client-side functions.[12]

In this code, each function body is wrapped inside an F# monadic block, using either `server` or `client` to identify type of the monad, `ClientM` or `ServerM` respectively. In all we make use of three monadic modalities in this paper:

Builder	**Monadic Type**	**Modality**
`client`	`ClientM<'a>`	Client-side code
`server`	`ServerM<'a>`	Server-side code
`client_async`	`ClientAsyncM<'a>`	Client-side non-blocking code

Using a typed solution is very appealing – thanks to the monadic syntax, the type of the server-side code is not compatible with the client-side code and vice-versa.

Note that monadic typing is not being used to write *pure* programs: in particular you can use regular F# programming side effects from both the client-side and server-side, as is always the case in ML programming. This is done by using the `do` and `do!` constructs of the monadic syntax discussed in §2. In essence monadic syntax and typing is used to specify a modality for mixed functional/imperative code. We are using monads to tame control modalities, not to tame side-effects.

This is partly because we can't expect that users will use only a fixed set of monadic side-effecting operations, if only because of the need to reuse F# and .NET standard libraries. Additionally, when an entire module or file has the same environment affinity the monadic syntax feels too verbose. Verification of the modal correctness of the code is further discussed in section §3.2.

The types of the three functions defined in Figure 5 are following:

```
TextChanged : obj * EventArgs -> ClientM<unit>
Process     : AstNode         -> ClientM<unit>
GenerateImg : string          -> ServerM<unit>
```

[12] All client-side members has to be annotated using `ReflectedDefinition` attribute, however we omit it in the examples, because it is not directly relevant for logic of our code.

Using the monadic bind operator (`do!` or `let!` in F#), it is possible to call client-side function from other client-side functions – as demonstrated in Figure 5, where function `TextChanged` calls function `Process` using the `do!` operator. Writing a code that tries to call client-side code from a server-side code causes a type mismatch, because in such code, the `do!` operator expects a value of type `ClientM<unit>`, but is given a value of `ServerM<unit>`. Therefore the following code fails to type-check:

```
member this.ClientCode () =
 client
  { -> () }
member this.ServerCode () =
 server
  { do! this.ClientCode() }
```

If the code is written in this way, the type system ensures that the calls between modal functions are correct.

This way of separating code that is executed in different environments is also extensible. We discuss several possible extensions later in future work and summary (§5).

2.3 Asynchronous Server Callbacks

Asynchronous calls back to the server from the client-side code are key aspects of Ajax applications. Traditionally in JavaScript these are implemented using events, so that the program registers a function to be called when response is received (and continues executing on the client-side). In this paper we call code that executes by de-scheduling at all blocking I/O operations and rescheduling via the event loop "asynchronous" code.

In the symbolic manipulation example we use asynchronous code to refresh the graph of the function. We want to implement the behavior so that the program checks for changes in the expression periodically, but never sends more than one request to the server and performs checking for the change with some delay (to prevent server overloading when typing an expression).

```
member this.RefreshImage(lastExpr) =
 client_async
  { do! Timer.SleepAsync(1000)
    let newExpr = this.txtExpr.Text
    if (lastExpr <> newExpr) then
      let! url = serverExecute
        (this.GenerateImg(newExpr))
      match url with
        | Some(u) ->
           do this.lDrawMsg.Text <- "Success"
           do this.imgGraph.ImageUrl <- u;
        | _ ->
           do this.lDrawMsg.Text <- "Failed!"
     do! this.RefreshImage(currentExpr); }
```

Figure 6. The function that checks for changes in the entered expression and refreshes the displayed function graph.

```
member this.Client_Load(sender, e) =
 client
  { do! asyncExecute(this.RefreshImage("")); }
```

Figure 7. Function that starts the refresh loop when page is loaded on the client-side.

Figure 6 shows a function that checks for the changes in the input field (accessed via `this.txtExpr`) and updates URL of image element on the page (`this.imgGraph`) and updates status label (`this.lDrawMsg`). To denote that the code is executed asynchronously, it is defined in a different monad type, identified by the `client_async` value. This monadic type is not compatible with the `ClientM<'a>` type mentioned earlier and so we need an explicit function call to execute it as demonstrated in Figure 7. The call to a server-side code also has to be done using an explicit function call which transforms monadic type. The two functions that are used in this example have following signatures:

```
asyncExecute  : ClientAsyncM<'a> -> ClientM<'a>
serverExecute : ServerM<'a> -> ClientM<'a>
```

The `do!` and `let!` operators in the `client_aysnc` block of course accept only other `ClientAsyncM` code, which allows us to write a recursive call at the end of the function (which itself has signature `string -> ClientAsyncM<unit>`). This typing property also prevents users from writing a code that would block the browser user interface, because calls back to the server can be done only from an asynchronous modality.

The `ClientAsyncM` monad is essentially an implementation of the continuation monad, where computations have type `(unit -> 'a) -> unit`[13]. This is implemented in F# code that is translated to JavaScript. This also means that it is possible to write primitive asynchronous functions as needed – to demonstrate this we show source code of the asynchronous sleep function which stops the execution for specified amount of time at Figure 8. The generated JavaScript code is demonstrated at Figure 9. The F# code isn't significantly shorter, but was much easier to write thanks to the static typing guarantees.

```
let SleepAsync(ms:int) : ClientAsyncM<unit> =
    PrimitiveStep (fun cont ->
        let t = new Timer();
        t.Interval <- ms;
        t.Elapsed.Add(fun (sender, e) ->
          t.Stop();
          cont() )
        t.Start(); )
```

Figure 8. F# source code for the SleepAsync function.

```
function SleepAsync(ms) {
  return PrimitiveStep(createDelegate(this,
    function (cont) {
      var t = CreateObject(new Timer(), []);
      t.set_Interval(ms);
      t.get_Elapsed().add(createDelegate(this,
        function (sender, e) {
          t.Stop();
          cont();  }));
      t.Start(); }));
}
```

Figure 9. JavaScript generated for the SleepAsync function.

When calling server-side function with arguments, or when the server-side function returns a value, the data needs to be serialized and sent over network. Aside from core F# types (tuples, records, lists, arrays, and algebraic data types) we also need to provide mechanisms for using certain types of objects that are used often in F# programming. This topic is further discussed in section §3.2 and §3.3.

3. Translation Techniques

The basic translation step is that we analyze the loaded application during the web request using reflection and determine what parts of the page should be executed on client side. We than read the quotation data of the appropriate parts of the application and translate them to JavaScript. The rest of the code that should be executed on server side is runs as a regular .NET program on the server side.

[13] This can be read as "a function that will generate an `'a` value sooner or later and till call the continuation when it's available"

3.1 Translating F# to JavaScript

The presented translator understands with a few exceptions all F# language constructs and also uses of standard F# types (algebraic data types, records, tuples, lists and arrays). Implementing support for most of the functional programming constructs used in F# in JavaScript was a relatively easy task, because JavaScript supports first-class functions and emulating basic algebraic data types and tuples using objects is straightforward. Additionally, no special care is needed to support list types, because lists are represented in terms of unions. To support sending of values from client-side code back to the server-side we need to preserve type information for all values, so we can deserialize the type correctly on the server-side.

There are however a few difficulties with the JavaScript language that we find interesting and that could be helpful for future "JavaScript" generators as well. The first is that JavaScript distinguishes between statements and expressions and so we need to find a way for generating JavaScript expressions from a code that produces JavaScript statement. Typical example of code that produces a statement is sequence of expressions ("`a; b`"). We also need to treat conditional expression with `unit` return type and conditional expression that return a value as distinct cases.

The second problem that we encountered is different variable scoping in JavaScript. According to the specification, the scope of any variable is the entire function from where the variable was defined. This can cause problems when translating a code where one variable is re-used during the execution, for example the index variable in a `for` loop. The following code creates an array of lambda functions in a for loop:

```
var f = [];
for(var i=0; i<10; i++)
{
  var x = i;
  f.push(function() { document.writeln(x); });
}
for(var j=0; j<10; j++)
  f[j]();
```

In JavaScript `i` is a mutable variable, and using it directly in a lambda function would capture it as an "l-value", i.e. a reference to the variable, however using `x` to store value locally in the `for` loop (as we did in the example) doesn't help – the scope of the `x` variable is the entire function body and value of `x` is mutated during execution. Our translator resolves this issue by generating a new JavaScript function to capture the variable scope as it is used in F#:

```
var f = [];
for(var i=0; i<10; i++) (function()
{
  var x = i;
  f.push(function() { document.writeln(x); });
})();
for(var j=0; j<10; j++)
  f[j]();
```

Next issue is that JavaScript doesn't support tailcalls. Two possible ways for supporting this are mentioned in [3]. One option is to use the JavaScript `setInterval` function which executes the given continuation in newly created context (after a specified time), the second option is to generate a trampoline (i.e., wrap a call in a loop and throw an exception with a continuation when depth reaches some level). The recursive function in Figure 6 used a `Timer.AsyncSleep`, which is internally using a `setInterval`, so it didn't suffer from this problem[14].

Since part of this work requires interoperation with the class-based object oriented ASP.NET framework, the translator supports F# object types as well. JavaScript uses prototype-based OOP, while F# uses class-based OOP style. To overcome this distinction, we use an ASP.NET AJAX JavaScript framework for class-based OOP simulation [7]. Since the code is generated automatically, this is sufficient solution, because developers don't have to learn how to use the class-based extensions.

3.2 JavaScript Mappings

When writing client-side applications we need to access to native JavaScript components that don't correspond to any F# or .NET library type, for example functions that manipulate the DOM[15]. To allow accessing to these functions from F# we first need to define a new mock type with functions of the right signatures and annotate it using attributes that define the JavaScript code that implement the function. The following code demonstrates how a mapping can be defined for the JavaScript `window.alert` function:

```
[<Mapping("window", MappingScope.Global,
          MappingType.Field)>]
type Window =
  [<Mapping("alert", MappingScope.Member,
            MappingType.Method)>]
  static member Alert (message:string) =
    (raise ClientSideScript:unit)
  ...
```

Figure 10. Mapping for the `window.alert` function.

The purpose of this mock type is just to provide a type-safe specification to use from client-side F# code. It is not expected that the code will be executed, so the body just raises an exception. Ideally it should be possible to verify during the compilation that the code will never be executed, the ways adopted in this work as well as possible improvements are described in §5.2.1.

The mapping in Figure 10 is defined using the .NET attribute `Mapping` which specifies the name in the target code, and the scope (which can be `Global` for global variables or functions or `Member` for accessing object members like methods or fields) and type of the target construct, which can have one of the values shown in Figure 11. During the translation process, all uses of functions annotated using this attribute will be replaced with the defined native call.

```
type MappingType =
| Method        // <inst>.Foo(<args>)
| Property      // <inst>.set_Foo/get_Foo(<value>)
| Field         // <inst>.Foo
| Object        // new Foo(<args>)
| Inline        // Foo(<args>)
```

Figure 11. Type that specifies target language construct for translation from F# to JavaScript.

3.3 External Types and Modules

The second type of mapping that the translator performs allows client-side code to use types and modules that exists in the F# or .NET library, but where these don't define JavaScript mappings using attributes. We want to allow using such types in the client-side code, because it removes the need to duplicate standard library types and makes programming much more uniform. Also some types and functions are very closely tied with the F# language (for example replacing standard `String` type would be very difficult).

In the symbolic manipulation we used a type displayed at Figure 12 to represent the internal state of the parser. The uses the .NET collection type `Stack<'a>` which is available in the client-side code thanks to the external type mappings, so we will use it to show how external mappings can be defined.

```
type Env =
  { Nodes:Stack<AstNode>;
    Operators:Stack<Operator>;
    lastIdent:bool; }
```

Figure 12. Type representing internal state of the parser.

The external mappings can be defined for types and modules from the standard F# and .NET libraries and are simply types/modules that have the same structure as the original types/modules, but which can be used on client-side, which means that it can consist of client-side F# code (when we need to re-implement the functionality) or mappings to native JavaScript components (when the same functionality already exists in JavaScript).

The example at Figure 13 demonstrates mappings for .NET `Stack<'a>` type, declared as a client-side type and for the `Int32` type that combines use of mapping to native JavaScript code and re-implementations of certain functions.

```
[<ClientSide; ExternalType(type Stack<obj>)>]
type Stack_CS<'a> =
  member this.Push(v:'a) =
    this.lst.Add(v)

  member this.Pop() =
    ...

[<ClientSide; ExternalType(type Int32)>]
type Int32_CS =
  [<Mapping("Lib.Convert.ToString",
      MappingScope.Global, MappingType.Method)>]
  member this.ToString() : string =
    (raise ClientSideScript)

  static member Parse(s:string) =
    ...
```

Figure 13. Mapping for .NET types `Stack<'a>` and `Int32`.

We found that the combination of client-side code translated to JavaScript, mappings to native JavaScript functionality and mappings to re-implemented functionality is a very powerful combination that allows us to define an entire client-side library for our project in a type-safe way using F# alone. Even advanced functional programming such as the monadic constructs mentioned earlier in §2.3 are written purely in F#.

3.4 Environment Separation

The environment (i.e. modality) where the code can be executed is determined either by a monad type (when using a monadic syntax) or by an attribute that specifies the environment explicitly. The type system ensures that calls between functions written using monads are correct; however for verifying the rest of the rules we decided to design an extension to the F# compiler, that checks if types are used in the right contexts, where context can be either a monadic block or a module/class annotated using a specific attribute. Implementation of this extension is further discussed in section §5.2.1.

The rules that need to be verified are following:

[14] In our current implementation we don't automatically generate any of the two possible options, but we plan to implement one of the outlined solutions.
[15] DOM (Document Object Model) represents a way for manipulating the displayed page and HTML elements in the browser.

- Standard F# types (tuples, algebraic data types, records, lists and arrays) are treated as *neutral* and can be used in both environments[16].
- F# and .NET library types and functions from modules are treated as *neutral* only when JavaScript mapping defined using external type mechanism described earlier exists, otherwise can be used as a *server-side* only.

Modules and classes defined by the user need to be annotated using one of the following attributes:

- **NeutralSide** – Code marked using this attribute will be treated as *neutral* and can call only other *neutral* code.
- **ClientSide** – The entire type/module can be used only at *client-side* and it can call only *neutral* or *client-side* code.
- **MixedSide** – Every member of the type/module marked using this attribute explicitly define the environment using monadic type (`client` or `server`), all members that are not annotated are treated as *server-side*.
- All other types and modules are treated as *server-side* only (and we still need to verify that they don't call any client-side only code).

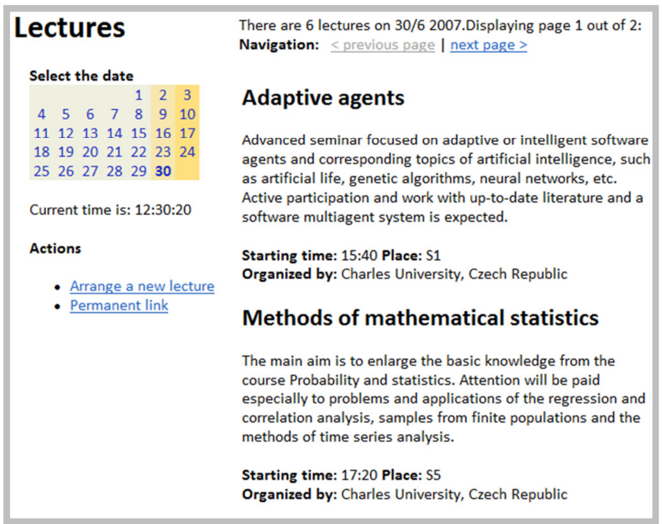
4. Example: Lecture Organizer Web

In the second example we focus on data driven web applications, by which we mean applications that display some data from the database using different views, allow users to edit the data and so on. We use our project to develop an application for planning lectures that has the following behavior:

- The web site contains a calendar where user can select a date and a list of lectures for the selected date. If the list contains more than specified number of lectures, the data spans across multiple pages.
- When the user selects a different date in the calendar the first page with lectures for the selected date is loaded, without reloading the entire page.
- When user clicks on the "next" or "previous" button, the displayed data change (without reloading the page) and the label with information about current page is updated.

Figure 14. Screenshot of the lecture organizer sample application.

The page is composed from two parts. HTML markup defines the overall look of the page and instantiates controls from which the page is composed (calendar, data listing, etc…). The second part is F# source code that defines page logic and interaction between the controls. Part of the source for the lecture organizer example is displayed at Figure 15.



Lectures

There are 6 lectures on 30/6 2007.Displaying page 1 out of 2:
Navigation: < previous page | next page >

Select the date

				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	**30**	

Current time is: 12:30:20

Actions

- Arrange a new lecture
- Permanent link

Adaptive agents

Advanced seminar focused on adaptive or intelligent software agents and corresponding topics of artificial intelligence, such as artificial life, genetic algorithms, neural networks, etc. Active participation and work with up-to-date literature and a software multiagent system is expected.

Starting time: 15:40 **Place:** S1
Organized by: Charles University, Czech Republic

Methods of mathematical statistics

The main aim is to enlarge the basic knowledge from the course Probability and statistics. Attention will be paid especially to problems and applications of the regression and correlation analysis, samples from finite populations and the methods of time series analysis.

Starting time: 17:20 **Place:** S5
Organized by: Charles University, Czech Republic

```
[<MixedSide>]
type Meetings =
  inherit ClientPage

  [<DuplexField>]
  val mutable selPage : int
  val calDate : Calendar
  val listLectures : Repeater
  val imgWait : Image

  member this.UpdateData () =
   server
    { let dt = this.calDate.SelectedDate
      let ds = Db.LoadPage(dt, this.selPage)
      do! this.listLectures.SetData(ds) }

  member this.NextPage (sender, e) =
   client
    { do  this.selPage <- this.selPage + 1;
      do  this.imgWait.Visible <- true;
      do! asyncExecute
           (client_async
             { do! serverExecute(this.UpdateData())
               do  this.imgWait.Visible <- false }}
  ...
```

Figure 15. Code that loads lectures for the next page.

4.1 State Management

In this section we explain how the code in Figure 15 executes, but first let us shortly explain what motivates the implementation. One of the goals of our work is to make it possible to compose the application from several independent components, because it allows users to develop controls that can be easily reused in multiple applications. The developers of the controls will typically want to expose some functionality that can be limited to a specific environment (server or client side).

In the earlier example with symbolic manipulations, the integration between client-side and server-side was implemented explicitly – the client-side code called a function on the server-side and processed the returned results, but for developing controls we require a slightly different semantics. When some functionality of the control is invoked from the server-side code of the page, we want it to behave like a self-contained operation, but in the case of pure functions we would have to collect all results and invoke controls after returning to the client-side again to update its visual representation. Alternatively the execution control could be transferred between server and client-side during the execution, but in the case of complex server-side code involving updates of many controls, this would lead to poor performance.

In the presented implementation, state management is another aspect of the server monad, which allows controls to record state changes that should be performed on the client-side. We can see where this occurs in the Figure 15, when we look for uses of the `do!` operator, which represents monadic bind and so can be used for accumulating state changes that will be sent to the client-side. In the sample code it is used only when calling `SetData` to set the displayed list of lectures and indeed this is the only place where state needs to be collected in this example.

When the `NextPage` function calls a server-side instance method of the page (`UpdateData`) using `serverExecute`, the object that represents page is created on the server-side including all members that represent controls (for example `calDate` for the calendar) and values of all fields marked using `DuplexField` attribute are sent from the client as well (in this example we need to access index of the selected page from both server and client sides). Than the code in the server monad is executed, collecting all changes to the state of particular controls and finally the state changes together with the function return value are sent as a response back to the client, which applies all the collected state changes to the client-side state.

The use of a primitive operation that collects a single state change is show at Figure 16.

```
member this.SetData(data) =
 server
  { do! „(§this).set_ClientData(§data)“ }

member this.set_ClientData(data) =
 client
  { let html = (* ... generate html ... *)
    do  this.InnerHtml <- html; }
```

Figure 16. Implementation of the `SetData` member in the Repeater.

The implementation of the Unicode double quotation mark operator[17] („ ... “) uses a quotation template literal (the compiled representation of the F# quotations as described in [10]) with spliced values to capture the essence of the operation to be performed. As we already described in §1.2, the operator (§) splices a value of the expression in the quotation tree, which means that in the example at Figure 16 we get a tree representing a call to the `set_ClientData` function with a spliced value referencing the control and a spliced value referencing the data as an arguments. Using this information the operator produces a server monad value which represents the invocation and when the execution of the server side code completes, the client side function `set_ClientData` is called.

5. Summary

5.1 Related Work

From our point of view there are two problems that need to be investigated in the web programming. The first problem, which is the complexity of writing client-side code, can be solved by replacing JavaScript altogether, extending JavaScript as a language or by providing a compiler from another language to JavaScript. The extensions to the language can be written as a set of functions that capture common programming patterns, as in [7] which contains a layer for emulating .NET programming patterns. The compiler to JavaScript can take a low-level language as an input (like Java bytecode in [5] or .NET IL in [4]), in which case it can become very complex, or it can take high-level language (for example C# in [6] or Links language in [3]). Another promising approach to the development of the client side code was presented in [26], which enables use of the functional reactive programming style for describing client side behavior in a language based on JavaScript, which is compiled to an ordinary JavaScript code.

The second problem is the integration between client and server-side code. There are many attempts to make it possible to use the same programming language for writing client and server-side code, but integrating code for both sides in one program is more difficult problem and there are only a few projects that attempt to solve it (one of them is the Links project [3]).

There are a several projects that try to solve some of the web development problems mentioned in this paper. The following table shows a summary of several possible approaches and the projects that follow them. Our project is displayed under a code-name **F# Web Tools** as well.

[17] Alternatively we also allow `<@! ... !@>` syntax.

	Client Language	**Client Runtime**	**Server Language**	**Integrated Code**
MS AJAX	JS[‡]	JS	Any	no
Script#	C#	JS	Any	no
GWT	Java/bytecode[†]	JS	Any	no
Volta	C#/VB/.NET[†]	JS	Any	no
Silverlight	.NET	.NET	Any	no
Links	Links	JS	Links	yes
F# WT	F#	JS	F#, ASP	Yes

Figure 17. Comparison of several approaches to web programming.
[†] Language support depends on completeness of the decompiler from low-level code (bytecode in case of GWT or IL in case of Volta).
[‡] With several extensions that emulate .NET programming patterns.

Commercially used frameworks based on OO principles focus mostly on server-side component-based development and providing a way for describing client-side interactions between components. In ASP.NET AJAX [7] or Backbase [12] this can be done by specifying declarative descriptions of the interactions using XML that are processed by a native JavaScript engine which is part of the framework. Declarative definitions are easier to write and can be also easily verified for correctness to some point, but are limited in what interactions they allow users to define and require knowledge of domain specific (XML based) language that is specific for every web framework.

The second problem that we deal with is the language impedance mismatch. Solving this problem in the web-development field is one of the main goals of the Links project [3, 17] where the Links language is compiled and executed differently when running on client, server and when accessing the database. The translation to JavaScript can be done from a high-level programming language as in Links project (which takes the functional Links language as a source) and in Script# [6] (which takes C# as a source language) or from a low-level language. Low level language translation from Java bytecode is performed by Google Web Toolkit [5] and from the .NET IL by Volta [4]. The overall complexity of our implementation stack is very low in comparison to these approaches: our entire translator and library mappings consist of approximately 3,000 lines of F# code and only a handful of lines of bespoke JavaScript. This has convinced us that the foundational elements offered by F# combine to give by far the best environment for applied heterogeneous execution of this kind, regardless of whether F# Web Tools becomes the world's biggest web-development platform or not.

In the .NET environment, abstractions for data access were investigated in [18] and [19] and are being implemented commercially in [20]. In F# the data-access without a language impedance mismatch is presented in [10].

The integration between code executed on the client-side and server-side in our work is tight, but the calls between different execution environments are explicit. A very interesting approach for the integration of sides is used in Links [3], where the entire code is compiled to a continuation passing style code and calls different environments are allowed implicitly. Another project that allows tight integration between execution environments is the HOP language [21].

Other related projects that focus mostly on the language impedance mismatch (like [4, 5, 6]) don't provide any direct integration between client-side and server-side code, but allow using RPC or Web Services for invoking server-code from the client side.

Another aspect of web programming is the separation between web design aspects (CSS, HTML, etc…) of the page and the application logic (e.g., F# code). Our implementation is based on ASP.NET [9] where this separation is directly supported. The advantages of separating application logic from HTML markup are also discussed in [22], but its implementation in projects that integrate client and server-side code is not very common and most of the related projects [3, 5, 21] work with HTML markup directly from the language, however [17] discusses several possible abstractions in the future work section.

A different approach for simplifying development of the client-side code is taken in the Silverlight project [15], which requires installation of a web-browser plug-in instead of producing JavaScript code. We can see Silverlight as another execution environment, because it is able to execute .NET IL code and supports a limited set of standard libraries, but since it doesn't provide any mechanism for integrating client and server-side code it might be interesting to extend our project to support Silverlight as another target client-side environment.

5.2 Future Work

5.2.1 Verification using Compiler Extensions

In the section §3.4 we informally described a rules that have to be verified in order to ensure that the environment separation of the program is correct, even in the parts that are not written using monadic typing.

To verify these rules we are working on a compiler extension for F#[18], which will allow code checking mechanisms to be added to the F# compiler. In general we think that compiler extensibility would be very helpful in any meta-programming scenario where the code is executed in heterogeneous environments and where additional *ad hoc* restrictions exist, for example in the examples presented in [10] (when translating subset of F# code to SQL or when executing subset of F# on GPU).

5.2.2 Web Development Issues

There are also several issues related to the web development that we'd like to address in the future. Writing code for the client-side in F# is significantly easier than writing code in JavaScript, however debugging client-side code is still difficult. An interesting solution for this problem is used in [5], which provides a debugging environment where the client-side code is executed natively instead of translating it and executing it in the browser and so the code can be debugged using any debugging tools for Java. A similar solution would be applicable to our project.

We also find very interesting an approach used for execution control in [3], which makes it possible to call a client-side code from a server-side thanks to the use of continuation-passing style. Adopting similar techniques in our work should be possible thanks to use of the F# monadic syntax, where execution control can be defined by a monad.

5.3 Conclusions

In this paper we have shown how F# (or any other appropriately extended version of ML) can be used to tackle three of the key issues in client/server web programming: the heterogeneous nature of execution, the discontinuity between client and server parts of execution and the lack of type-checked execution on the client side. We use F# meta-programming to serve the client-side portions of an F# application as JavaScript, which makes it possible to write programs running in web browsers in a type-checked functional language without installing any extensions to the browsers. The meta-programming is non-intrusive through the use of attributes to indicate client-side and server-side portions of the application.

[18] We expect to release an initial version of the extensibility mechanism for F# compiler and extension for verifying separation of environments prior to the workshop.

We also demonstrate mechanisms for accessing native JavaScript functionality from F#, which together with the translator gives us enough expressive power to write an entire client-side library purely in F#. We use monadic modalities in F# to separate the code intended to run in specific environments and, thanks to the typing properties of monads, we make calls between different environments explicit, which prevents users from writing incorrect code and also gives a clue where application performs inefficient call between environments.

In the presented work we used these two techniques together to allow development of web applications with both client and server-side functionality written in an integrated way in a single language, but the same approach for separating environments could also be as used in cases where all parts of the application execute natively.

From a web development perspective, we allow writing client-side code in an asynchronous way similarly to the continuation-passing style presented in [17], but we make this explicit using F# monadic syntax. We also presented a way for building composable components that expose separate client-side and server-side functionality as part of their interface.

Acknowledgments

The project was developed during a Microsoft Research Internship and we thank to everyone who made it possible. Special thanks to James Margetson for his useful comments and suggestions.

References

[1] Jesse James Garrett. Ajax: A new approach to web applications. Adaptive path, 2005

[2] Ali Mesbah; Arie van Deursen. An Architectural Style for Ajax. In *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture*, 2006

[3] Ezra Cooper, Sam Lindley, Philip Wadler, Jeremy Yallop. Links project. The Links website, 2007. See http://groups.inf.ed.ac.uk/links/

[4] Erik Meijer et al. Project Volta. Microsoft, 2007. See http://channel9.msdn.com/ShowPost.aspx?PostID=223865

[5] Google Web Toolkit. The GWT website, 2007. See http://code.google.com/webtoolkit/

[6] Nikhil Kothari. Script#. The Script# website, 2007. See http://projects.nikhilk.net/Projects/ScriptSharp.aspx

[7] ASP.NET AJAX. Microsoft, 2007. See http://ajax.asp.net/

[8] Don Syme and James Margetson. The F# website, 2006. See http://research.microsoft.com/fsharp/.

[9] ASP.NET. Microsoft, 2007. See http://asp.net/

[10] Don Syme. Leveraging .NET meta-programming components from F#: Integrated queries and interoperable heterogeneous execution. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 2006.

[11] Don Syme, Gregory Neverov, James Margetson. Extensible Pattern Matching via a Lightweight Language. To appear in *Proceedings of the Inter-national Conference on Functional Programming (ICFP '07)*. ACM, 2007

[12] Backbase AJAX Solutions. The Backbase website, 2007. See http://www.backbase.com/

[13] Tim O'Reilly. What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. O'Reilly Media, Inc, 2007

[14] ECMAScript Language Specification. ECMA 262 3rd edition, 1999.

[15] Silverlight. Microsoft 2007. See http://silverlight.net/

[16] Tomas Petricek. F# WebTools project website, 2007. See http://tomasp.net/fswebtools

[17] Ezra Cooper, Sam Lindley, Philip Wadler, Jeremy Yallop. Links: Web Programming Without Tiers, In *Proceedings of 5th International Symposium on Formal Methods for Components and Objects '06*. 2006

[18] Gavin Bierman, Erik Meijer,Wolfram Schulte. Programming with rectangles, triangles, and circles. XML Conference, 2003.

[19] Gavin Bierman, Erik Meijer,Wolfram Schulte. The essence of data access in Cω. In *Proceedings on the 19th European Conference on Object Oriented Programming*, pages 287–311, July 2005.

[20] Microsoft Corporation. The LINQ May 2006 Preview, 2006. See http://msdn.microsoft.com/data/ref/linq/

[21] Manuel Serrano, Erick Gallesio, and Florian Loitsch. HOP, a language for programming the Web 2.0. Proceedings of the First Dynamic Languages Symposium, Portland, Oregon, October 2006.

[22] David L. Atkins, Thomas Ball, Glenn Bruns and Kenneth C. Cox. Mawl: A domain-specific language for formbased services. Software Engineering, 25(3):334 346, 1999.

[23] Robert Pickering. Foundations of F# (book). Apress, 2007. ISBN 978-1590597576

[24] Don Syme, Adam Granicz, Antonio Cisternino. Expert F# (book). Apress 2007. ISBN 978-1590598504

[25] Mono. Mono Project website See http://www.mono-project.com

[26] Leo Meyerovich. Flapjax: Functional Reactive Web Programming. See http://www.cs.brown.edu/~lmeyerov/thesis8.pdf