Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Tomáš Petříček

Client side scripting using meta-programming

Department of Software Engineering

Advisor: RNDr. David Bednárek

Study program: Computer Science, Programming

2007

Thanks to Don Syme for his enormous support of this project and also for his work on F# that made the project possible in a form in which it is presented in this thesis. Also thanks to members of the Links project: Ezra Cooper, Philip Wadler and Jeremy Yallop for an insightful discussion and to James Margetson and Simon Peyton Jones for their useful comments and suggestions. Finally, thanks to my advisor RNDr. David Bednárek, for his patience and for supervising of this work.

I hereby certify that I wrote the thesis myself using only the referenced sources.
I agree with lending the thesis.

Prague, 2 August, 2007 Tomáš Petříček

Contents

Title: Client side scripting using meta-programming
Author: Tomáš Petříček
Department: Department of Software Engineering
Supervisor: RNDr. David Bednárek
Supervisor's e-mail address: david.bednarek@mff.cuni.cz

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 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 work we present the first project that deals with all three mentioned problems but which still integrates with existing web 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 makes it possible to 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 type defining the web page logic. The 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.

Keywords: ML; F#; Meta-programming; Ajax; Web Development; Language Impedance Mismatch

Název práce: Client side scripting using meta-programming
Autor: Tomáš Petříček
Katedra (ústav): Katedra Softwarového Inženýrství
Vedoucí bakalářské práce: RNDr. David Bednárek
e-mail vedoucího: david.bednarek@mff.cuni.cz

Abstrakt: Webové aplikace založené na principech souhrnně označovaných zkratkou „Ajax“ se dnes stávají de-facto standardem, ale vývoj takovýchto aplikací je naneštěstí velmi náročný. Vývojáři musí čelit problémům, jako je nesourodost programovacích jazyků v různých vrstvách aplikace, omezené běhové prostředí v prohlížeči a chybějící integrace mezi serverovou a klientskou částí aplikace, které musí být psány jako dvě nezávislé části, ale obvykle tvoří jednu homogenní aplikaci. V této práci představujeme první projekt, který usiluje o řešení všech tří zmíněných problémů a současně umožňuje použití stávajících technologií jako je ASP.NET na straně serveru a JavaScript na straně klienta. Používáme jazyk F# pro psaní obou částí aplikace, což umožňuje vyvíjet klientskou část typově bezpečným způsobem s použitím části F# knihoven, dále umožňujeme propojení obou částí aplikace v jednom homogenním typu, který určuje logiku aplikace. Při spuštění je aplikace vykonávána různorodě, část pomocí JavaScriptu v prohlížeči a část jako nativní kód na serveru. K oddělení serverového a klientského kódu používáme monadický zápis jazyka F# a tím se toto rozlišení stává vlastností typového systému.

Klíčová slova: ML; F#; Meta-programování; Ajax; Webový vývoj; Language Impedance Mismatch

Preface

During early phase of this work I had a chance to discuss a design of the project with Don Syme from Microsoft Research, who first suggested using the F# language designed by him and its meta-programming capabilities for solving one of the issues targeted by this work.

This collaboration led to my internship in Microsoft Research in Cambridge, where I worked under the mentorship of Don Syme on an integrated client/server framework for web development, which is now released as F# Web Toolkit[1] under the Microsoft Permissive License.

The original intention of this work was to prove that the meta-programming approach is sensible however thanks to the increasing quality and importance of the F# language it was possible to implement a solution useful in practice. In addition the recent enhancements in the F# language allowed solving many interesting related problems of web development that are presented in this thesis.

Finally, the project was presented to the Links team at University of Edinburgh and to the Programming Principles and Tools group at Microsoft Research who both provided many useful comments and suggestions and some of them are mentioned in future work section of this thesis.

[1] The project is available at: http://www.codeplex.com/fswebtools

Chapter 1
Introduction

Our goal in this work is to build a framework for developing interactive web applications, which is a task that occupy the lives of the thousands of programmers who currently write web applications in JavaScript (on the client-side) and PHP/C#/VB.NET/Java/Ruby (on the server-side). This class of web applications can, broadly speaking, be called Ajax[2] applications. This work shows how the F# language [8] 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. The use of F# for developing both sides of the application also makes it possible to develop richer client-side code, possibly leveraging of the functional language constructs available in the F# language.

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. While an interaction with the Ajax application is driven by the client-side, the application is authored on the server and served to the client in response to the 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.

We show how to solve three of the fundamental difficulties of the 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. The work is built in 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 [24] akin to that of Haskell. On the server side we use ASP.NET framework [9], because it is easily accessible from F#, though we believe that the presented approach could be easily adapted to be used with many other web development tools. We also discuss what language features are crucial for supporting particular features of the presented work.

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

From a language point of view, we present the following interesting aspects:

- We use meta-programming to write a code that is executed across heterogeneous environments using a single homogenous language. As far as we are aware, we are the first to use this approach for a web development scenario.
- We present the first translator from F# (an ML-family functional language in general) to JavaScript, including the translation of the F# core language constructs as well as a subset of the F# and .NET libraries. We also show how native JavaScript components[3] can be accessed from the source language in a type-safe way.
- We also discuss approaches 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 a single file (and in the case of user interface components even in a single class) 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 using F# "asynchronous workflows" in the 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 the mechanism for managing client-side state changes performed by server-side components during asynchronous callbacks to the server.
- We also 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.

1.1 What Makes Web Applications Hard

The first reason that makes developing web applications a difficult task 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.

[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)

In general, the ability to execute homogenous application in a heterogeneous environment is becoming more and more important, not only in the web development scenario – there are several projects that integrate data-access in the language [18-20, 10], there is a number of attempts to integrate the writing of general purpose GPU programs [27, 10] and finally in the web development scenario, the heterogeneous execution is to some extent used in [3, 4]. We believe that this observation changes the whole way we should think about designing compilation hierarchies and tool support for programming 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 (in the web development scenario for example [4, 5, 6, 30]), but only a few projects go beyond this and deal also with the separation of client and server-side code (mainly Links [3] and partially also [4]).

Another problem appears when using server-side frameworks such as ASP.NET, Ruby on Rails 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 on the server. Reusable components (as available for example in ASP.NET) can hide complex client-side functionality (for example the calendar control can update itself on the client-side using JavaScript to display different month), however on the client-side every component behaves like a black-box and there is no way to specify client-side interactions, which are an important aspect of any interactive web application. For example, a calendar control may need to interact with a data list control to filter information (already loaded on the client-side) according to the selected date, which is an interaction that should not involve "going back to the server" to rebuild the entire page, as is required by ASP.NET. Achieving this kind of "smoothness" is partly a reason 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, which is the goal of our work.

The rest of the work is structured as follows – In the 0 we introduce the F# language, especially focusing on parts that are relevant for our F# to JavaScript translator and language features that are frequently used when writing code using the F# Web Toolkit. This chapter also contains quick overview of the state of the art in the web development. Then in Chapter 3 we identify key problems of existing web development technologies that we approach in this work. In Chapter 4 we briefly demonstrate our solution to these problems in case-studies presenting three real-world sample applications developed using our project.

[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.

After this short overview, the following Chapter 5 discusses possible alternative solutions to several problems and also explains reasons for choosing the solution we implemented. In Chapter 6 we discuss our implementation in a larger detail. This discussion includes the translator from the F# language to JavaScript (§6.1), a way for building richer client-side environment (§6.2), our solution for integrating client and server-side code (§6.3), the support for data types that can be used in both execution environments (§6.4) and finally our approach for writing component-based composable code (§6.5). Since our solution is based on a general purpose programming language we believe it is important to discuss what language constructs (especially those not common in many mainstream languages) enable implementing a solution like ours and so this discussion is presented in Chapter 7. Finally, Chapter 8 gives overview of related work, possible extensions to our project that we would like to implement in the future and a conclusion.

Chapter 2
Background

In this section we first briefly discuss the F# language [8]. In order to make this thesis self contained, we also give a brief overview of the core F# types, F# language constructs that support several programming paradigms (functional, imperative and object-oriented) and also the interoperability with the .NET platform – since the translator from the F# language to JavaScript will be presented later in this work, we find it important to describe at least informally what does the F# language consist of and what has to be considered when working on such translator.

We also discuss the F# language features that are intensively used when developing applications using the F# Web Toolkit, especially 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 and we also informally describe a recent addition to F#: the monadic syntax akin to that of Haskell, which is very important for our work.

Finally, we introduce a web development problem in general and changes in the understanding of web caused by a rise of Ajax based interactive web applications, together with a brief overview of the existing web development frameworks relevant to the presented work.

2.1 F# Language and Runtime

The F# language is documented on the F# web site [8] and in two books [23, 24]. In [10] the following description is given: "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."

2.1.1 Functional programming in F#

F# is a typed language, by which we mean that types of all values are known during the compile-time. Thanks to the use a type inference, the types are explicitly specified in the code very rarely. Basic data types (aside from a standard set of primitive numeric and textual types) available in F# are tuple, discriminated union, record, array, list, function and object. In the following quick overview, we use F# interactive, which is a tool that compiles and executes the entered text on the fly.

Overview of F# Data Types
The first example demonstrates constructing and deconstructing tuple type:

```
> let tuple = (42, "Hello world!");;
val tuple : int * string

> let (num, str) = tuple;;
val num : int
val str : string
```

The syntax used for deconstructing the value into variables `num` and `str` is in general called pattern matching and it is used very often in the F# language – the aim of pattern matching is to allow matching value against a pattern that specifies different view of the data type – in case of tuple, one view is a single value (of type tuple) and the second view is pair of two values (of different types). F# also supports generalized pattern matching constructs called *active patterns*, which are discussed later in this overview. In the next sample we demonstrate working with the discriminated union type:

```
> type Expr =
  | Binary   of string * Expr * Expr
  | Variable of string
  | Constant of int;;
(...)

> let v = Binary("+", Variable "x", Constant 10);;
val v : Expr
```

To work with the values of a discriminated union type, we use pattern matching again. In this case we use the `match` language construct, which can be used for testing a value against several possible patterns – in case of the `Expr` type, the possible options are `Binary`, `Variable` or `Constant` expressions. The following example declares function `eval`, which evaluates the given expression:

```
> let rec eval x =
    match x with
    | Binary(op, l, r) ->
        let (lv, rv) = (eval l, eval r)
        if (op = "+") then lv + rv
        else failwith "Unknonw operator!"
    | Variable(var) ->
        getVariableValue var
    | Constant(n) ->
        n;;
val eval : Expr -> int
```

Further, the record type can be viewed as a tuple with named members, which can be accessed using a dot-notation:

```
> type Product =
    { Name:string;
      Price:int; };;
(...)

> let p = { Name="Test"; Price=42; };;
val p : Product

> p.Name;;
val it : string = "Test"
```

The types used for storing collections of values are list and array. F# list is a typical linked-list type known from many functional languages – it can be either an empty list (written as `[]`) or a cell containing a value and a reference to the tail (written as `value::tail`). Array is a .NET compatible mutable array type, which is stored in a continuous memory location and is therefore very efficient – being a mutable type, array is often used in imperative programming style, which is discussed later. The following example shows declaration of a list value and an implementation of a recursive function that adds all members of the list:

```
> let nums = [1; 2; 3; 4; 5];;
val nums : list<int>

> let rec sum list =
    match list with
    | h::tail -> (sum tail) + h
    | [] -> 0
val sum : list<int> -> int
```

The important feature when writing recursive functions in F# is the support for *tailcalls* – meaning that when a last operation performed by the function is a call to a function (including a recursive call), it drops the current stack frame and minimizes a chance for getting a stack overflow exception. The sum function from the previous example can be written using a tail recursion as following:

```
> let rec sumAux acc list =
    match list with
    | h::tail -> sumAux (acc + h) tail
    | [] -> acc
val sum : int -> list<int> -> int

> let sum list = sumAux 0 list
val sum : list<int> -> int
```

The next F# type is a function – in F#, as in many other functional languages, functions are first-class values, meaning that the function can be given as an argument to other functions and also returned from a function (a function that takes function as an argument or returns function as a result is called *high-order* function). The important aspect of working with functions in functional languages is the ability to create *closures* – creating a function that captures some values available in the current stack frame. The following example demonstrates a function that creates a function for adding specified number to an integer:

```
> let createAdder n =
    (fun arg -> n + arg);;
val createAdder : int -> int -> int

> let add10 = createAdder 10;;
val add10 : int -> int

> add10 32;;
val it : int = 42
```

The type of the result (`int -> int -> int`) denotes that when the function is called with `int` as an argument, it produces a value of type function (which takes integer as a parameter and produces integer as a result). In fact, the previous example could be simplified, because any function taking more arguments is treated as a function that produces a function value when it is given the first argument, which means that the following code snippet has the same behavior (note that the types of the function `createAdder` declared earlier and the type of the function `add` are the same):

```
> let add a b = a + b;;
val add : int -> int -> int

> let add10 = add 10;;
val add10 : int -> int
```

Many functions in the F# library are implemented as high-order functions. For example standard set of functions for manipulating with list values is demonstrated in the following example:

```
> let odds = List.filter (fun n -> n%2 = 0) [1; 2; 3; 4; 5];;
val odds : list<int> = [1; 3; 5]

> let squares = List.map (fun n -> n * n) odds;;
val squares : list<int> = [1; 9; 25]
```

Expressions and Variable Scoping

The F# language doesn't have a different notion of a statement and an expression, which means that every language construct is an expression with a known return type. If the construct performs only a side effect and doesn't return any value, the type of the construct is `unit`, which is a type with one possible value (written as "`()`"). The semicolon symbol (;) is used for sequencing multiple expressions, but the first expression in the sequence is required to have a `unit` as a result type. The following example demonstrates how the `if` construct can be used as an expression in F# (however in the optional F# *lightweight syntax*, which makes whitespace significant, the semicolon symbol can be omitted):

```
> let n = 1
  let res =
    if n = 1 then
      printfn "n is one";
      "one"
    else
      "something else";;
n is one
val res : string = "one"
```

Unlike some languages that allow one variable name to appear only once in the entire function body (e.g. C#) or even treat all variables declared inside the body of a function as a variable with scope of the whole function (e.g. Visual Basic or JavaScript), the scope of F# values (the name value is used instead of a variable, because F# variables are immutable by default) is determined by the `let` binding and it is allowed to hide a variable by declaring a value with the same name:

```
> let n = 21
  let f =
    if n < 10 then
      let n = n * 2
      (fun () -> print_int n)
    else
      let n = n / 2
      (fun () -> print_int n)
  let n = 0
  f ();;
42
val it : unit
```

In this example, the value `n` declared inside a branch of the `if` expression is captured by a lambda function, which is returned from the `if` expression and bound to the value named `f`. When the `f` is invoked it indeed uses the value from the scope where it was created (in languages, where the variable named `n` would refer to a value stored globally, it would be rather problematic to write a sample like this).

2.1.2 Imperative programming in F#

Similarly as ML and OCaml, F# adopts *eager evaluation* mechanism, which makes it semantically reasonable to support imperative programming features in a functional language. By default, F# value bindings are immutable, so to make them mutable the `mutable` keyword has to be used. Additionally F# supports a few imperative language constructs (like `for` and `while`), which are expressions of type `unit`:

```
> let n = 10
  let mutable res = 1
  for n = 2 to n do
    res <- res * n
  res;;
val it : int = 3628800
```

The use of the eager evaluation and the ability to use mutable values makes it very easy to interoperate with other .NET languages, which is an important aspect of the F# language.

2.1.3 .NET interoperability

The .NET BCL[5] is built in an object oriented way, so the ability to work with existing classes is essential for the interoperability. Many (in fact almost all) of the classes are also mutable, so the eager evaluation and the support for side-effects are next two key features when working with the .NET library. The following example demonstrates working with the mutable generic `List<T>` type from the BCL (note that we use underscore as a type argument, which is possible when the type inference can deduce the type argument automatically):

[5] BCL stands for "Base Class Library" and includes all classes available in the default installation of the .NET Framework.

```
> open System.Collections.Generic
  let list = new List<_>()
  list.Add("hello")
  list.Add("world")
  list |> Seq.to_list;;
val it : string list = ["hello"; "world"]
```

As a fully compatible .NET language, F# also provides a way for declaring its own classes, which are compiled to .NET compatible class types and therefore the types can be accessed from any other .NET language as well as extend classes written in other languages. This is an important feature when we want to access .NET libraries like ASP.NET.

2.1.4 Object oriented programming

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 and subtyping. F# object types can have fields, constructors, methods and properties (a property is just a syntactic sugar for getter and setter methods). The following example introduces the F# syntax for object types:

```
type MyCell(n:int) =
  let mutable data = n + 1

  member x.Data
    with get()  = data
    and  set(v) = data <- v

  member x.Print() =
    printf "Data: %d" data

  static member FromInt(n) =
    MyCell(n)
```

The object type `MyCell` has a field called `data`, property called `Data`, an instance method `Print`, a static method `FromInt` and an implicit constructor that initializes the value of the field. The declaration of an interface (called abstract object type in F#) is similar:

```
type AnyCell =
  abstract Value : int with get, set
  abstract Print : unit -> unit
```

The interesting concept in the F# object oriented support is that it is not needed to specify explicitly whether the object type is abstract (interface), concrete (class) or partially implemented (class with abstract methods), because the F# infers this automatically depending on the members of the type.

F# also supports type upcasts (`o :> TargetType`), downcasts (`o :?> TargetType`) and dynamic type tests (`o :?> TargetType`), which tests whether a value can be casted to specified type. Additionally, all F# data types are subtypes of the `obj` type, which is equivalent to the .NET `Object` type.

2.1.5 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 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 and can be also used to extract abstract syntax trees of a members discovered using System.Reflection (note that the F# “quotations” are a feature of the F# compiler and as such can’t be used with programs produced from C# or VB).

F# and .NET Reflection Library

The F# library also extends the .NET System.Reflection to give additional information about F# data types – for example we can use the reflection library to examine possible values of the `Expr` type (discriminated union) declared earlier:

```
> let exprTy = typeof<Expr>
  match Type.GetInfo(exprTy) with
  | SumType(opts) -> List.map fst opts
  | _ -> [];;
val it : string list = ["Binary"; "Variable"; "Constant"]
```

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 declaring `Documentation` attribute (simply by inheriting from the `System.Attribute` base class) and adding it to a static method in a class:

```
type DocumentationAttribute(doc:string) =
  inherit System.Attribute()
  member x.Doc = doc

type Demo =
  [<Documentation("Adds one to a given number")>]
  static member AddOne x = x + 1
```

Using the .NET System.Reflection library it is possible to examine members of the `Demo` type including reading of the associated attributes (which are stored in the compiled DLL and are available at run-time):

```
> let ty = typeof<Demo>
  let mi = ty.GetMethod("AddOne")
  let at = mi.GetCustomAttributes
                (typeof<DocumentationAttribute>, false)
  (at.[0] :?> DocumentationAttribute).Doc;;
val it : string = "Adds one to a given number"
```

F# Quotations

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 and explicitly mark piece of code as a quotation and the second way is to use

ReflectedDefinition attribute, which instructs the compiler to store quotation data for a specified top-level member. The following example demonstrates a few simple quoted F# expressions – the quoted expressions are ordinary type-checked F# expressions wrapped between the Unicode symbols « and »[6]:

```
> « 1 + 1 »
val it : Expr<int>

> « (fun x -> x + 1) »
val it : Expr<int -> int>
```

Quotation processing is usually done on the *raw* representation of the quotations, which is represented by the non-generic Expr type (however the type information about the quoted expression is still available dynamically via the Type property). The following example implements a trivial evaluator for the quotations (GenericTopDefnApp pattern matches with the use of the plus operator, the Int32 pattern recognizes a constant of type int):

```
> let plusOp = « (+) »
  let rec eval x =
    match x with
    | GenericTopDefnApp plusOp.Raw (_, [l; r]) ->
       (eval l) + (eval r)
    | Int32(n) ->
       n
    | _ ->
       failwith "unknonw construct"
val eval : Expr -> int

> let tst = « (1+2) + (3+4) »
  eval tst.Raw
val it : int = 10
```

Figure 1. Using pattern matching for quotation processing

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[7] (the |> is a pipelining operator, which applies the argument on the left hand side to the function on the right hand side):

```
> « §db.Customers
      |> filter (fun x -> x.City = "London")
      |> map (fun x -> x.Name) »
val it : Expr<Seq<string>>
```

In the raw representation, the spliced value can be recognized using the LiftedValue pattern, which returns a value of type obj, which can contain any F# value.

[6] Alternatively, it is also possible to use <@ and @> symbols.

[7] This example is based on the FLinq project, which integrates data-access to the F# language using meta-programming. The FLinq project is presented in [10].

The second option for quoting F# code (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. We refer to this option as a non-intrusive meta-programming, because it allows processing of the member body (e.g. translating it to JavaScript), but doesn't require any deep understanding of meta-programming from the user of the library. The following code gives a simple example:

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

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

Extensible Pattern Matching via Active Patterns

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# quotation terms. We already used one variant of active patterns in earlier example in Figure 1, where `Int32` and `GenericTopDefnApp` were both active patters.

Active patterns are just functions which a special name and are extremely useful when implementing a quotation processing code, because active patterns can be used to group similar cases together. In the following example we declare active pattern that recognizes two binary operations:

```
let (|BinaryOp|_|) x =
  match x with
  | GenericTopDefnApp plusOp.Raw  (_, [l; r]) -> Some("+", l, r)
  | GenericTopDefnApp minusOp.Raw (_, [l; r]) -> Some("-", l, r)
  | _ -> None

let rec eval x =
  match x with
  | BinaryOp (op, l, r) ->
      if (op = "+") then
        (eval l) + (eval r)
      else
        (eval l) - (eval r)
  (* ... *)
```

The name `(|abc|_|)` indicates that the matching may succeed and return a value denoted by the pattern `abc` or may fail. Other types of active pattern can be found in [11].

2.1.6 F# Computation Expression

The last feature of F# that is important for our work is the F# monadic syntax (also called *computation expressions*), 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 least informally, because we rely on it in some parts of our project.

Properties of a monadic type are defined by a builder type which specifies a 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 Return : 'a -> M<'a>
  member Delay  : (unit -> M<'a>) -> M<'a>
  member Let    : 'a * ('a -> M<'b>) -> M<'b>
```

The `Bind` and `Return` 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). The `Delay` member allows building monads that are executed lazily.

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

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

Figure 2. Sample computation written using monadic builder `mb`.

Here, `mb` is an instance of type `MBuilder`[8] and `mFunc` is a function with signature `unit -> M<int>`. When using an 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 the monadic builder:

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

Monadic syntax in F# allows embedding of "non-standard" computations in the language, for example the continuation monad can be used for writing programs that execute asynchronously (possibly without blocking the main application thread). The computation written using the monadic syntax has a type determined by the monadic builder (e.g. the type of the expression on a Figure 2

[8] 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.

is `M<int>`), which prevents the user from mixing computations in an incorrect way. It is however possible to "lift" an ordinary F# computation to a monad using the `let` and `do` binders. In the following example we give a sample implementation of the builder for a continuation monad (the value in the continuation monad is a function, which when receives a continuation as an argument, calculates the value and executes the continuation with the result of the computation as an argument):

```
type Cont<'a> = ('a -> unit) -> unit

let bind v d = fun cont -> v (fun res -> (d res) cont)
let result v = fun cont -> cont v
let delay  f = fun cont -> (f ()) cont

type ContinuationBuilder() =
  member x.Bind(v, d) = bind v d
  member x.Return(v)  = result v
  member x.Delay(f)   = delay f
  member x.Let(v, f)  = bind (result v) f

let cont = ContinuationBuilder()
```

In the presented work we make use of the following constructs from the monadic syntax (the syntax of the expression in the monadic block (`cexp`) is similar to the ordinary F# expressions (`exp`) and it also provides similar language constructs):

```
exp =
  | ident { cexp }
  | ...

cexp =
  | let! pat = exp in cexp        monadic bind (mb.Bind)
  | let pat = exp in cexp         regular bind (mb.Let)
  | do! exp                       monadic "unit" bind (mb.Bind)
  | do exp                        regular "unit" bind (mb.Let)
  | return! exp                   monadic return (expr)
  | return exp                    regular return (mb.Return)
  | if exp then cexp1 else cexp2  conditional (if)
  | if exp then cexp              "unit" conditional (if)
  | match exp with (pat -> cexp)+ pattern match
  | for pat in exp do cexp        for-each loop over expr
  | while exp do cexp             while loop
  | try cexp finally cexp         try-finally block
  | try cexp with (pat -> cexp)+  try-catch with exception 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.

2.2 Web Development

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 the immediate interaction with the user and combine the behavior of desktop applications and traditional web applications.

Developing highly interactive applications is a difficult task, partly because this requirement wasn't considered 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 update 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 presented project is independent to this specific technology and it could be easily adapted to use a different model.

2.2.1 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 of applications than the particular technology that is used to implement that control flow.

The differences between control flows are demonstrated at the following two diagrams, which represent initial request and one page update in a traditional web and Ajax based applications.

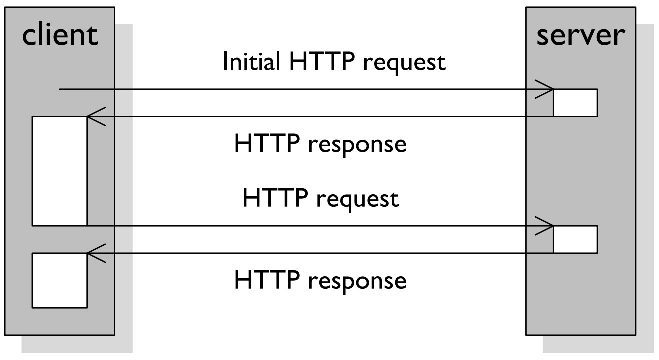


Diagram 1. Control flow of traditional web application

In a 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.

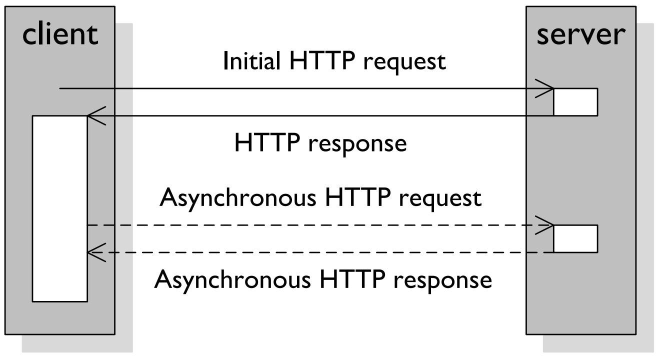


Diagram 2. Control flow of interactive "Ajax" application

In an Ajax application (Diagram 2), the client initiates with a request to the server, but the answer from server consists of data and also a 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.

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.

2.3 Web Development and Frameworks

In this section we give a brief overview of existing web development platforms used in practice as well as interesting research projects. We discuss possibilities for building richer client-side environment, frameworks used to build client-side Ajax applications and technologies that are used to build the server-side code. In general the web development tools can be divided into the following groups:

- The application is primarily a server-side application and generates HTML with embedded client-side code that is executed to perform the client-side behavior.
- The application is primarily a client-side application and accesses the server-side functionality via web services.
- Frameworks that go beyond the traditional client/server separation and provide integrated development environment to some point.

Indeed, the first type of applications is currently the most frequent; however the separation between these groups is often very blurry, because several approaches are often mixed in somewhat confusing ways.

2.3.1 Client-Side Languages

A typical requirement for a web application is to work correctly in all frequently used web browsers and platforms, which limits the choice of a client-side programming languages to JavaScript[9]. JavaScript is an interpreted language with runtime type-checking and a support for prototype based object-oriented programming. However with the increasing importance of the complex client-side applications, there is a need for supporting other programming paradigms than those adopted by JavaScript.

There are essentially three different ways for dealing with this need. First way is extending the JavaScript runtime by implementing an emulation layer for different paradigms in JavaScript (e.g. class-based object oriented programming in [7] or functional reactive programming in a part of [26]). The second way is compiling different language to JavaScript, be it a high level language (for example in [3, 5, 6, 26]) or a low level bytecode in [4]. Finally, the third approach is abandoning JavaScript altogether and replace it by a different, richer client-side runtime environment, as for example project Silverlight [15].

We believe that JavaScript is important as a target runtime, and we adopt the earlier mentioned view of JavaScript as an "assembly language" for the web, however we keep in mind that in the future different client-side runtime environment may become important, so in the presented work, JavaScript is used as one of the possible targets.

2.3.2 Client-Side Frameworks

The limitations of the client-side code mentioned earlier make it difficult to define reasonable abstractions for client-side frameworks. Some of the commercially used frameworks like ASP.NET AJAX [7] or Backbase [12] provide way for declarative description of the interactions using XML that is processed by a JavaScript engine which is part of the framework. Declarative definitions are easier however allow expressing only limited set of interactions and are hardly extensible.

To allow richer interactions on the client-side ASP.NET AJAX [7] allows developers to write interaction code also in JavaScript (aside from the declarative XML language). Since this framework is focused on .NET developers, it provides set of objects and functions that emulate class-based OOP in JavaScript with stronger runtime type checking based on class-based properties of the objects. This makes the code to some extent less fragile, but requires learning of JavaScript with specific language extensions that may sometimes feel inhomogeneous.

[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.

Promising approach is used in Flapjax [26], which provides a client-side functional reactive programming model, where the code can be either written in JavaScript or in a Flapjax language that is compiled to JavaScript. The advantage of functional reactive programming is that it provides a declarative, but fully extensible way for writing the client-side code, meaning that there are no limitations of the expressiveness as in XML based approaches.

2.3.3 Server-Side Languages and Frameworks

The traditional approach for writing a server-side code allowed embedding short scripts that were executed on the server-side when serving the HTML documents. This programming style has many problematic aspects (the most important is difficult maintainability and reusability of the code) and is not suitable for building complex applications. It is important to note that in any server-side framework, the goal of the server-side application is to produce HTML (possibly with JavaScript code to define client-side functionality) that will be sent to the client as a response.

In general there are two orthogonal aspects of the server-side development. The first is providing a way for writing server-side applications using a hierarchical structuring of the code and the second specifies the way interactions are encoded.

The first aspect determines how the developers create assets that can be re-used in multiple applications and how these assets are composed. For example the page (which is usually the top-level unit) is responsible for overall page-related properties of the application and it delegates specific parts of the application (e.g. menu, controls for modifying the view properties or the part that presents the data) to a specific asset, be it a control in object-oriented frameworks or a function or set of functions in tools based on functional programming style. To give a few examples, the functional style is used in [21] and the object oriented style in [9, 29, 36]. The asset has to expose some way for controlling its properties, and is also responsible for interactions inside the asset (e.g. calendar is responsible for highlighting the selected date) and for generating the part of the response (HTML and JavaScript output of the asset).

The second aspect is a way for specifying the interaction between the components (e.g. when a user selects a date in the calendar, display the data according to the selected date). In the OO world, the event-driven paradigm is very common and is usually implemented by some form of MVC (model-view-controller) design pattern (this approach is adopted in Ruby on Rails [29] and in a less pure form also in ASP.NET [9]). We believe that more declarative way for solving this problem would be useful, however as far as we are aware this is possible only in a very limited form in ASP.NET data-view controls.

The most interesting contribution to the second aspect from the world of functional programming is a concept of continuation based web frameworks (e.g. [31]). In these frameworks the applications can be written without the inversion of control [28], which means that the control flow that specifies the interactions is encoded in the code in a linear way and the code is executed as a long-running application whose state is serialized to some permanent store when the page is sent to the client. This way of programming has some appealing properties (e.g. the code is more readable), but has several problematic aspects (e.g. requires complex session handling and it loses the notion of addressability, which is very important for web applications).

2.3.4 Integrated Web Development Tools

The first step in integrating client and server sides is to allow writing both sides in a single language (this includes using some of the approaches for extending client-side runtime environment described earlier in 2.3.1), but without the tight integration between the sides, which means that the calls from the client to the server are made explicitly using some form of remote procedure calls (usually using the `XmlHttpRequest` object in JavaScript). This approach is used in Google Web Toolkit [5], which uses Java language and in haXe [30], which has its own language.

Volta [4] goes slightly further, because it provides two execution modes – in the *debug* mode, the entire application executes in the Volta browser as a single desktop-like application, to produce a *release* version of the application (which is executed heterogeneously, part on the client as a JavaScript and part on the server), the application has to be modified by adding explicit cross-side calls, though this process is partly automatic via refactoring tools. Volta also allows developers to use in theory[10] any .NET language, because it translates low-level .NET IL code.

Finally the Links project [3] provides a way for writing fully integrated client-server code in the functional Links language. The project focuses on the language impedance mismatch and also provides data-access constructs directly in the language. It uses the continuation based model, in a way that the Links language compiler compiles the entire application using a continuation passing style, which makes it possible to perform calls between the server and client in both directions[11].

[10] „In theory", because the .NET IL is fairly complex and some .NET languages rely on a features that may not be supported by Volta or on a language-specific libraries that may not be translatable to the JavaScript. Anyway it supports at least mainstream .NET languages including C# and VB.NET.

[11] The call from server to client is possible because if the code is translated to a continuation passing style, the call is wrapped in to a continuation and is sent to the client as a response.

Chapter 3
Problem description

3.1 Language Impedance Mismatch

The typical web application consist of code that works with database (typically SQL), code that runs on the server side and generates the web page (C#, Java, PHP...) and script that runs on client-side and performs asynchronous callbacks and page updates (JavaScript). This means that the single developer needs to be able to write code in 3 significantly different programming languages. If we also count HTML (markup language describing the content) and CSS (language that describes the web page design) than we get even 5 different languages. For developing AJAX applications a very good knowledge of at least a language for writing server side code (C#, Java, PHP…) and the client side scripting language (JavaScript) is needed, but JavaScript is a dynamically typed language with prototype-based OOP features, while C# and Java are statically typed with class-based OOP[12], which means that the languages differ in the two most basic concepts and so mastering both of them is extremely difficult task.

For that reason the first goal of our project is to solve the language impedance mismatch by allowing writing the entire application in a single language (in our case F#). Fortunately, we don't have to have to cover the data-access integration in our work, because this is already available in [10] (building on top of the earlier work done in this area [20, 19, 18]), so our project focuses on providing a way for executing F# code on client-side using JavaScript.

3.2 Richer Client-Side Environment

This section could be also called "Framework impedance mismatch" akin to the name used in the previous one, because this term would summarize the essence of the second goal of our work. Broadly speaking, there are two problems that have to be solved, once we use the F# language for writing the client-side code. The first is how we can make the native JavaScript components accessible to the F# code and the second problem is how we can make the F# core functionality available on the client-side.

Both of the problems are very important – the first one because the client-side library is relatively complex and we need a simple way for accessing it, with the possibility to adjust parts of the library so the functionality visible to the F# user follows common F# programming practices. Part of this problem is also the fact that there are several differences between JavaScript library implementations,

[12]As described in [34]

especially in the DOM[13] objects. This problem can be either ignored (by giving access only to the compatible subset of features) solved by our framework (it would automatically decide what version of the code to use), or solved at the library layer (the library would dynamically switch between implementations, depending on the available functionality).

The second problem is important for the users of our toolset – we don't want to make it difficult to use our tools and thus we want to allow users to leverage of the F# and .NET knowledge that they already have, which also includes using basic F# functions and .NET types. This means that our solution has to provide a way for giving client-side implementation to the existing F# modules (containing basic functions) and .NET classes from the BCL.

3.2.1 Asynchronous Client-Side Programming

Another goal of allowing richer client-side environment is to provide a simple way for writing client-side code that is executed asynchronously, meaning that it can be executed without blocking the browser user interface or possibly other executing "thread". JavaScript doesn't support threads (as known from .NET or Java), but some operations are by nature non-blocking and are used very often in the client-side programming, which makes it important to support easy sequencing of these operations (examples of such operations are asynchronous call to the server, waiting using a timer or waiting for a specified event).

In the F# language, asynchronous operations can be written using the monadic syntax described earlier in §2.1.6. The following example (adopted from [24]) demonstrates the use of `async` monad for doing asynchronous I/O operations (the `use!` operator has the same meaning as `let!`, but is used for working with resources that have to be explicitly freed):

```
async
  { use! inStream  = File.OpenRead("Image.tmp")
    let! pixels    = inStream.ReadAsync(numPixels)
    let  pixels'   = TransformImage(pixels)
    use! outStream = File.OpenWrite("Image.done")
    do!  outStream.WriteAsync(pixels') }
```

This programming style is very appealing and is much more compact than the usual style using callback functions, so allowing similar style on the client-side is a very appealing goal.

3.3 Bridging the Client/Server Gap

In the problem description we so far focused on the development of the client-side part. Our project doesn't aspire to make any significant changes in the way the server-side part is written, so our next goal is to bridge the huge gap between the code that runs on the server and code that runs on the client.

[13] DOM stands for Document Object Model and is standard way for manipulating with the user interface elements displayed in the web browser, unfortunately not all browsers follow the W3C standards and some of the features that are provided by most of the browsers aren't specified by the standards.

Let us quickly recapitulate the situations where the gap is crossed in typical Ajax application – during the initial request the server-side code is executed and as a result, web site with the script representing the client-side code is sent back to the client and executed on the client-side. The client code can asynchronously invoke the server while still running on the client-side, wait for the results and update the client-side state (we'll refer to this as a *callback*), or it can force the browser to request other web page, which means that the execution of the client-side script will be stopped and control will be transferred to the server, with possibly sending a data representing the state produced on the client-side (we'll refer to this as a *postback*).

In server-side frameworks that represent page using controls, like in ASP.NET the control tree is recreated during every postback, so during initial request or every postback, the tree is built on the server, updated by the server-side code and then used to generate the response including the serialized state information[14]. Callbacks are typically done as static method invocations, which means that the method can't access, nor modify the page state, so it is not needed to rebuild the control tree during a callback. Handling of a callback therefore requires complex client-side code that processes the result from the server call and modifies objects representing the page state and visual representation on the client-side.

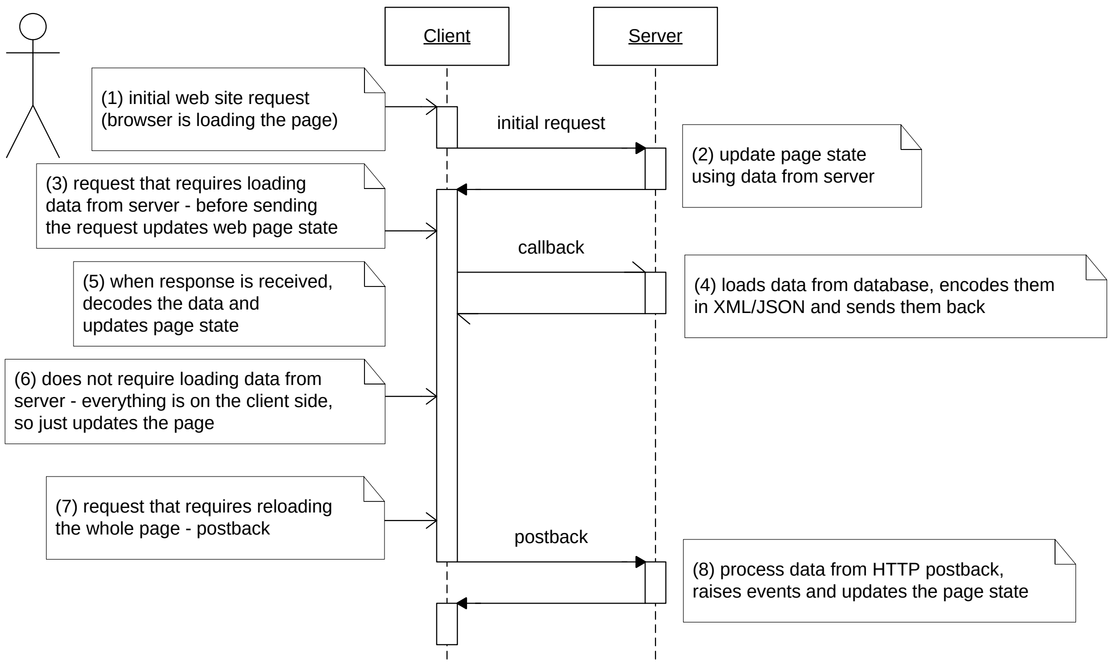

Client
Server
(1) initial web site request (browser is loading the page)
initial request
(2) update page state using data from server
(3) request that requires loading data from server - before sending the request updates web page state
callback
(4) loads data from database, encodes them in XML/JSON and sends them back
(5) when response is received, decodes the data and updates page state
(6) does not require loading data from server - everything is on the client side, so just updates the page
(7) request that requires reloading the whole page - postback
postback
(8) process data from HTTP postback, raises events and updates the page state

Diagram 3 – Workflow of example "AJAX" web application

Sample workflow showing interaction between user and "AJAX" based web application is displayed at the Diagram 3. The diagram shows 4 different kinds of situations. During the initial request (1) the whole page is generated at the

[14] In ASP.NET the serialized state is called "ViewState" and is stored in a hidden HTML input.

server side, then a request (3) requires loading data from server, so the client updates the page to notify the user that it is processing the request and invokes callback to the server. The server encodes data in JSON or XML format[15] and sends the response to the client, which needs to parse the encoded data and update the page state (again on the client side), later client requires displaying the data that were already downloaded to the client (6) so the client-side script just updates the page to display the data. Finally, the client requires loading of significantly different content (7), so the client invokes postback. After processing the HTTP request, rising of appropriate events on the server-side and updating the state on the server side, the page is rendered to (X)HTML and sent back to the client.

From the description above, you can see that workflow of Ajax based web applications can be very complex – indeed, everything could be processed uniformly on the server-side using postbacks, but this means reloading the page after every user's action, so the user experience would be really bad.

The biggest issue arising from the described workflow is the need to update the displayed page and its state after performing a callback, mainly because the code performing the update has to be very often duplicated in a server and a client-side code. In the presented project the aim is to make this duplication only at the lowest possible level (e.g. it is obvious that when changing the text displayed by a label, different code has to be executed on the client and server side, but a code in a page that sets the text can be essentially the same). Also we want to avoid the need for writing explicit updates of the client-side state after performing a callback. Some effort is also needed to develop the encoding and decoding of the data during callbacks and to define what data types can be safely sent across the client-server border.

3.4 Client/Server Components

Finally, since we're building our toolkit on top of ASP.NET it is essential to provide seamless integration with the ASP.NET control-based composition model. Currently, the control wraps the server-side behavior of some reusable building block and exposes properties that allow configuration of such control as well as events that are used for notifying the users of the control about a change in the control caused usually by an interaction with the user. Our goal is to extend this model and enable a development of controls that wrap both server and client-side behavior and expose both server and client-side properties and events. This essentially mixes two different aspects of the control in one self-contained asset and as far as we are aware this integration is not possible in any web framework available currently.

[15] In fact there are no limitations of the format that can be used, however XML (extensible markup language) and JSON (JavaScript object notation) are the most common, mainly because they can be relatively easily generated and processed on the client-side.

Chapter 4
Case Studies

The first overall decision that has to be made is whether the work should focus on building a project that can be easily used in practice, for instance by extending existing well-established technology and by allowing some non-verifiable, but practically useful constructs in the code, or a project that gives a pure, verifiable solution, but doesn't integrate with existing solutions or limits user from using certain problematic techniques, which may be useful in some situations.

We believe that the web development world gives more attention to projects that present a "proof by example" than a formal proof and so our goal is to find a reasonable trade-off and build a project that is as pure as possible, but still integrated with existing technologies, so it can be used to easily build non-trivial applications by using the existing developer knowledge of the platform and also by accessing some existing components. Let us therefore give a brief "proof by example" which will also introduce all important aspects of our work in this chapter.

In the first example (Windows Scripting using WSH) we will demonstrates the JavaScript translation including accessing native JavaScript functionality and allowing the use of .NET BCL classes in the JavaScript code. The second example (Web Symbolic Manipulation) introduces the web development framework by developing a rich client-side applications and also demonstrates integration between the client and server side. Finally, the last example (Lecture Organizer) demonstrates development of a typical control-based data-driven web application.

4.1 Case Study: Windows Scripting using WSH

Windows Script Host (WSH) is a scripting infrastructure for Windows that use JavaScript (or VBScript) for writing scripts that manipulate with Windows registry, file system, etc. The Figure 3 demonstrates an example code that can be written using our tools to write an F# script that asks user to enter an executable name and executes the executable. The F# code is indeed never executed – it is translated to JavaScript and executed using a WSH execution engine (`cscript.exe`).

The code is interesting for two reasons. First it manipulates with a shell objects that are available from WSH (`Shell` for launching a process and the `ShellProcess` for reading status of the process) as with ordinary F# types and second, it uses `Console` class (namely `ReadLine` and `WriteLine` methods), which is a .NET BCL type and so isn't available in the WSH, which provides its own I/O functions.

```
open System
open WshMappings

[<ClientSide>]
type SampleClass =

  [<ReflectedDefinition>]
  member this.Read () =
    let pn = Console.ReadLine()
    if pn <> "" then Some(pn) else None

  [<ReflectedDefinition>]
  member this.Main () =
    Console.WriteLine("Enter an executable name:")
    match this.Read() with
    | Some(name) ->
        Console.WriteLine("Starting '" + name + "' ...")
        let proc = Shell.Current.Exec(name)
        while (proc.Status = 0) do
          Script.Sleep(100)
        Console.WriteLine("Process completed with code: " +
                          proc.Status.ToString())
    | _ ->
        Console.WriteLine("No name entered!")
```

Figure 3. WSH script written in F# that demonstrates using BCL classes, accessing native functionality and a few basic F# language constructs and types.

To support the first case, where it is needed to provide mappings for a type that doesn't correspond to any existing BCL type we define a new mock type with the corresponding F# signature, but with no implementation and use a .NET attributes to give annotations that specify native code that implements the functionality (we refer to this type of mapping as an *internal mapping*).

```
type ShellProcess =
  [<Mapping("Status", MappingScope.Member, MappingType.Field)>]
  member x.Status : int =
    raise NativeCode

type Shell =
  [<Mapping("Exec", MappingScope.Member, MappingType.Method)>]
  member x.Exec(s:string) : ShellProcess =
    raise NativeCode

  [<Mapping("WScript.CreateObject(\"WScript.Shell\")",
            MappingScope.Global, MappingType.Inline)>]
  static member Current : Shell =
    raise NativeCode
```

Figure 4. Declaration of type-safe mappings for WSH objects. In this sample we define type `Script` with static members `In` and `Out` and the type that represents input stream.

In the Figure 4 we used an F# class to describe interfaces of two native JavaScript objects that were used earlier in the example in Figure 3. The purpose of this type is just to provide a type-safe specification that can be used from the F# code. The code will never execute, so the body only raises an exception. In this example, it is easy to ensure that the code is never executed (since the entire code is translated to JavaScript), but in a web scenario where translated code is mixed with the executed code, this problem becomes more important. Possible solutions are discussed later in §6.3.3.

The second type of mapping (called *external mappings*) is used when we want to allow working with a type that exists in the F# or .NET library, but doesn't define internal mappings using attributes. 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 are useable on client-side, meaning that they can consist of client-side F# code (when we need to re-implement the functionality) or internal mappings to native JavaScript components (when the same functionality already exists in JavaScript).

The example in Figure 5 demonstrates mappings for the .NET `Console` type, as we used it in the previous example. The methods of the `Console` type are in this case reimplemented using native types (declared using internal mappings).

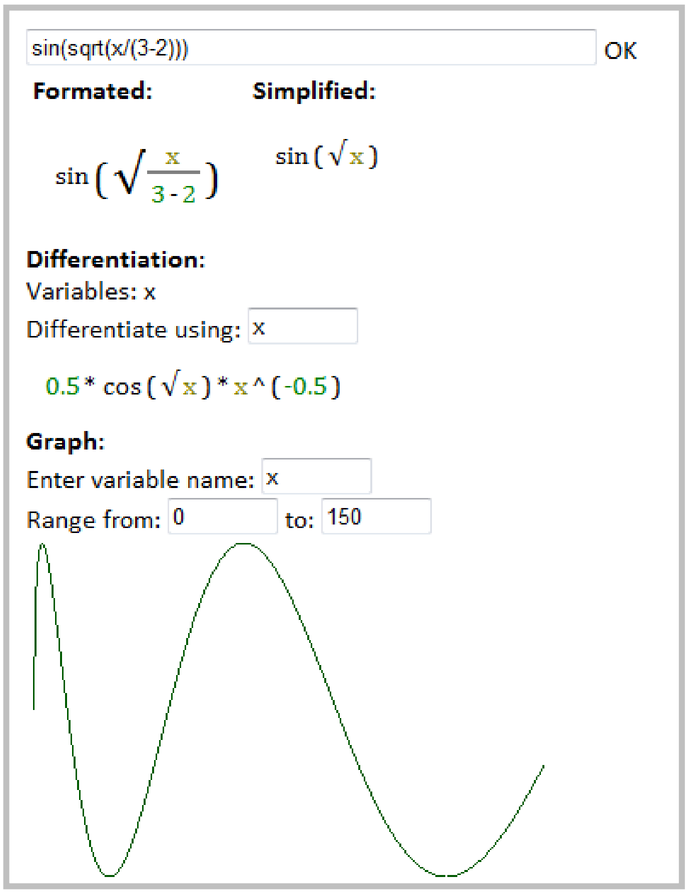
```
[<ClientSide; ExternalType(type System.Console)>]
type Console =
  [<ReflectedDefinition>]
  static member WriteLine(s) =
    Script.Out.WriteLine(s)

  [<ReflectedDefinition>]
  static member ReadLine() =
    Script.In.ReadLine()
```

Figure 5. Mapping for .NET type `System.Console`.

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 discussed in §6.3.2 are written purely in F#.

4.2 Case Study: Web Symbolic Manipulation

In the second example we use F# to develop an application running as a JavaScript code in a web browser, which performs tasks that are traditionally easy to solve in functional languages. 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 complete source code of the application is available in the Appendix and also at our web site [16], which also shows the running application. Figure 6 presents a screen-shot of the running application.


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

Variables: x

Differentiate using: x

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

Enter variable name: x

Range from: 0 to: 150

Figure 6. Symbolic manipulation code written in F# and running as a JavaScript code in the web browser

4.2.1 Symbolic Manipulation Functions

The parser and symbolic manipulation functionality 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 7.

```
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 7. Signatures of functions implementing the tokenization, parsing and symbolic manipulation in the sample as well as types used for representing the AST.

The module is marked using the `NeutralSide` attribute, which means that functions contained in it are implemented only using library functions and types that are available in both client and server 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 performing an I/O operations).

The following example shows a simple evaluation function (it is a simplified version of the function used in the actual example) written in F#:

```
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) in ea + eb
    | _ -> failwith "unknown"
  eval nd
```

The code has a few interesting aspects from the translator point of view – it uses higher order functions to read values of variables in the expression, it defines an inner recursive function and it is written using pattern matching on algebraic data type representing the AST. Our translator produces following code[16]:

```
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");
        } } }
  })
  return eval(nd);
}
```

Figure 8. JavaScript code generated from the F# `evaluate` function.

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

The technique used for translating F# to JavaScript is further described in section §6.1, where we mention all the problematic topics, like difference between expression and statement in JavaScript, different variable scoping etc.

4.2.2 Integrating Client and Server Code

As already mentioned, we wanted to permit writing a single class representing the behavior of a web application for both sides, where portions of the code run in multiple different environments. In this example, the page uses functions from the `Parsing` module described above and implements the following functionality:

- The entered expression is visualized using HTML (the visualization is being updated "live" as the expression is entered).
- The expression is simplified and symbolic differentiation is calculated as the expression is entered (updated "live" as well).
- 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 back to the client, which then displays the image.

First two operations involve executing only client-side code, 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 `GenerateImg` (which draws a graph of the function) is executed on the server-side. The functions are all members of the same object representing the page. Calls between the client and the server-side will be discussed shortly.

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

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

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

Figure 9. Example shows subset of page interaction logic of the symbolic manipulation application. It contains one server and two client-side functions.

In this code, each function body is wrapped inside an F# computation expression, using either server or client to identify type of the monad, `ClientM` or `ServerM` respectively (monadic types are more precisely described in §6.3). 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 (as in Haskell) and both sides can use regular F# programming with side effects using the `do` and `do!` constructs of the monadic syntax discussed in §2.1.6. We also allow using the `let` and `do` constructs (regular non-monadic bind operators) for lifting non-monadic computations in the monad, because this allows developers to access standard .NET and F# functionality and also use code which isn't written using the monadic syntax (e.g. using the `Parsing` module presented earlier).

The types of the three functions defined in Figure 9 are following:

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

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 9, where function `TextChanged` calls function `Process` using the `do!` operator. Nevertheless 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.

4.2.3 Asynchronous Server Calls

Asynchronous calls to the server from the client-side code are key aspects of Ajax applications. Traditionally in JavaScript these are implemented using events (which can be used from our project as well), but we tend to prefer a higher abstraction using F# monadic syntax and the `async` monad where possible.

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

Figure 10. Function that starts the refresh loop (using the `asyncExecute` primitive) when the page is loaded on the client-side.

```
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 11. The recursive function that checks for changes in the entered expression and refreshes the displayed function graph.

Figure 11 shows a function that checks for the changes in the input field (accessed via `this.txtExpr`), 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 cannot be directly called from the `ClientM<'a>` type (denoted by `client` value) mentioned earlier and so we need an explicit function call to execute it as demonstrated in Figure 10.

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>
```

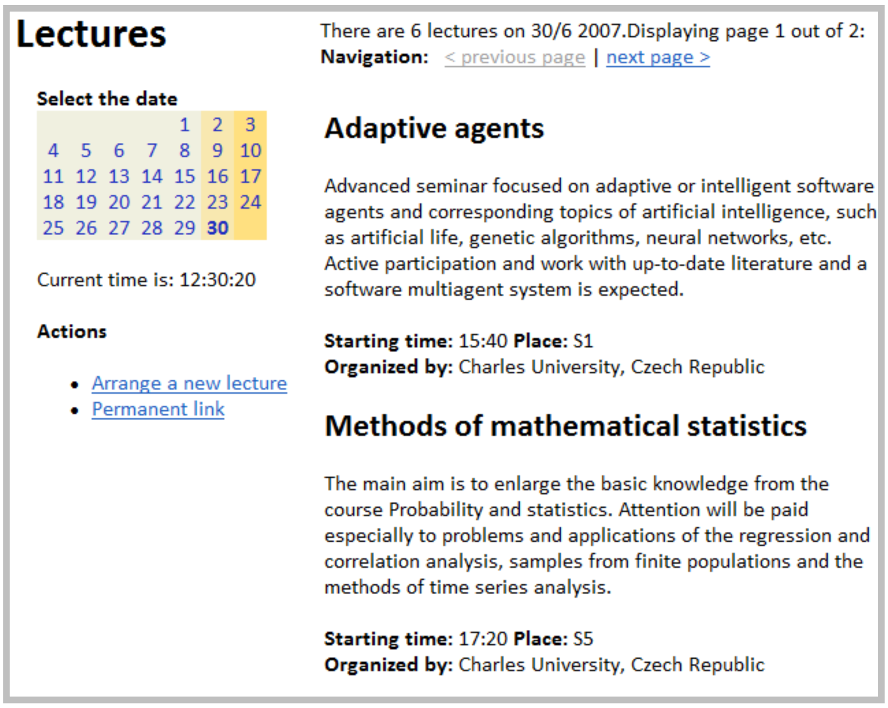
The do! and let! operators in the `client_aysnc` block of course accept other `ClientAsyncM<'a>` code, which allows us to write a recursive call at the end of the function (which itself has a signature `string -> ClientAsyncM<unit>`). This typing property also prevents users from writing a code that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

The conversion and explicit calls between monads raise several interesting questions; some of them are further discussed in §6.3. Also the implementation of the asynchronous operations (e.g. `Timer.SleepAsync`) is worth further discussion, because it is implemented purely in F# and such can be easily extended to support other primitive asynchronous operations (§0). Finally, when calling a server-side function with arguments, or when the server-side function returns a value, the data needs to be serialized and sent over the 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 §6.5.

4.3 Case Study: Lecture Organizer

In the last 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 with 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.


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

Figure 12. 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 the F# source code that defines page logic and interaction between the controls. Part of the F# source for the lecture organizer example is displayed in Figure 13. Some important aspects of the HTML markup will be discussed later.

```
[<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 13. Code that loads lectures for the next page.

4.3.1 State Management

In this section we explain how the code presented in Figure 13 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 13, when we look for uses of the `do!` operator, which represents the monadic bind operation 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 the `SetData` function 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, which is used in the implementation of the `Repeater` control is shown in Figure 14.

```
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 14. Implementation of the SetData member in the `Repeater` control.

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 §2.1.5, the operator (§) splices a value of the expression in the quotation tree, which means that in the example in Figure 14 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 argument. 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. This topic is further discussed in §6.5.

4.3.2 Data binding

Displaying data in ASP.NET uses a technique called *data-binding*. Using this technique it is possible to write a template (or several different templates) for displaying the data in the markup file and then instantiate the template by setting a data source. In ASP.NET this is indeed possible only on the server-side, so we provide an extension that enables using the same technique on both sides, which means that the setting data source of a control (e.g. `Repeater`) can be done from the server-side code (during the initial request) as well as from the client-side code after loading data using asynchronous call to the server.

[17] Alternatively we also allow using the ASCII (`<@! … !@>`) operator.

The syntax for writing templates is demonstrated in Figure 15. The sample shows markup declaring the `Repeater` control with the template for displaying one lecture (from the list of lectures that is displayed on the page). The code enclosed in the data binding markup tags (`<%# … %>`) accesses properties of the data type, which stores information about the lecture.

```
<fwc:Repeater id="listMeetings" runat="server">
<ItemTemplate>
<li>
  <h2><%# Container.RecdGet("Title") %></h2>
  <p><%# Container.RecdGet("Description") %></p>
  <p>
  <strong>Starting time: </strong>
    <%# Container.RecdGetFormat("DateAndTime", "hh:mm") %>
  <strong>Place: </strong>
    <%# Container.RecdGet("Place") %><br />
  <strong>Organized by: </strong>
    <%# Container.RecdGet("University") %>,
    <%# Container.RecdGet("Country") %></p>
</li>
</ItemTemplate>
</fwc:Repeater>
```

Figure 15. Declaration of the data binding template for the `Repeater` control.

Chapter 5
Discussion of Alternatives

In this chapter we discuss possible approaches to several aspects of our work as well as reasons for choosing the approaches that we implemented. We will first discuss the choice of the F# language and the use of JavaScript as a client-side execution environment together with alternative options for producing JavaScript code from a different language. We also discuss how the control flow can be controlled in an application like this and a few possible ways for integrating the client and the server-side code. Finally, we discuss the way the state is managed in our solution with a few other possible alternatives examined in the related work.

5.1 Language and Runtime

First of all, our goal was to use existing programming language, which has many practical advantages, though it requires having some non-standard way for distributing the code between the client-side and the server-side. Probably all languages can produce a native code that can be executed on the server-side, but the answer to a question what approach could be used for generating the client-side program, is an interesting problem.

The first concern regarding the client-side is what the target environment for executing the client-side code should be. Developing a custom plugin is not a choice for us, because we want to support as many platforms as possible. The remaining options are using JavaScript or using some existing, widespread plugin (for executing .NET code, the interesting option is the Silverlight platform). We believe that JavaScript is currently still the most important platform, but having in mind that this may eventually change, we try to design our work in a way that it could be easily adapted to target other platforms.

The decision to use JavaScript brings a question, how can we produce JavaScript from the language we want to use for writing the code (and how do we solve the language impedance mismatch)?

- First option is obviously writing a compiler from the selected language to JavaScript, but this is problematic because it duplicates many non-trivial tasks (e.g. type checking and type inference in a language like F#).
- The second option is to use low-level language as a source for the compiler, like Java bytecode or .NET IL code. This option is far more appealing, because it in theory allows using any language compiled to the chosen low-level code, however implementing translator is far more complicated (especially when the goal is to support any language producing the code)

and using this way we also lose many useful information about the code that could be used by the translator[18].

- The next option is using a DSEL (*domain specific embedded language*), which is a language expressed in terms of the host language. This approach was successfully used in several Haskell projects, because it requires using a language that allows overriding the semantics of many basic constructs, but even in such language it is more suitable for modeling languages of limited size than for providing an alternative execution model of the entire language.
- Finally, the last option, which we decided to adopt in our work, is using intensional meta-programming[19]. The advantage of this approach is that it is relatively easy to implement and it gives us a consistent way for writing meta-programs (all properties of the ordinary F# programs at the source code level also hold for the meta-programs). On the other side, the drawback is that this option relies on the meta-programming support in the language that we use – in case of F# this is possible thanks to non-intrusive meta-programming described in §2.1.5.

The main reason for using F# as a language is that it provides very good support for non-intrusive meta-programming, which is a key feature that allows us to use the same general purpose language for writing both a code that is executed natively and a code that is executed as a meta-program. As a .NET language, F# also nicely integrates with existing .NET web development technology (ASP.NET), which allows us to build a toolkit in a way that it will be familiar to many web developers and finally the recent addition of monadic syntax gives us a very interesting way for integrating the client-side and the server-side code as well as for expressing other non-standard computations (namely the asynchronous client-side code).

5.2 Control Flow Model

A fundamental design decision is what control flow model will be used in the framework. By a control flow, we mean how the application will react to external events and how we will encode the transition between client and the server when integrating both sides of the application.

- The event-based model is used in many object-oriented frameworks including ASP.NET. It provides very good control over how things are executed and is also familiar to many developers. It can be however criticized for its “inversion of control” and the need to maintain a global state in many situations.

[18] In case of IL and the F# language we couldn’t translate F# lambda functions to JavaScript functions, because IL doesn’t have notion of lambda function. Also the use of monadic syntax would be problematic, because we translate monadic syntax used in the client-side code to ordinary (non-monadic) JavaScript code.

[19] The term „intensional meta-programming“ is defined in [10] as “systems where the equational properties of the program are not preserved through meta-programming”, which is also our case where the code is executed in different environment.

- An alternative used in functional languages (e.g. [28]) is using the continuation passing style. This approach unfortunately hides some important aspects of the code and also limits the programming model (for example there is only one logical thread of execution, while in web application we often need to perform some action on the client-side while executing asynchronous call to the server).
- Functional reactive programming (for example [33]) gives a very appealing way for writing reactive programs, which web applications indeed are. FRP was also successfully applied to client-side web programming in [26]. The reason for not using FRP-based approach in our work is that we want to give a system that will be familiar to existing web developers, in particular to ASP.NET developers. Nevertheless we would like to examine possibilities for using FRP-based techniques in the future §7.5.3.

In our work we use primary the event-based model, although the asynchronous client-side code written using the `client_async` monad (a continuation monad) makes it possible to go a bit further and adopt, at least on the client-side, programming styles similar to processes from Links [3] and Actor based models described in [32]. These models use essentially a recursive function which receives messages representing the monitored event and calls itself with the accordingly modified state.

5.3 Client-Server Integration

The integration between the client-side and the server-side code in general purpose language is to some point innovative, because integrating two equally important execution environments in a single general purpose language is not common, thus the problem how we could represent the separation arises.

Aside from the use of monads which was demonstrated earlier in Figure 9 we also considered using .NET attributes as can be seen in Figure 16.

```
[<RunAt(Side.Client)>]
member this.TextChanged (s:obj, e:EventArgs) =
  let tok = Parsing.Tokenize(this.txtInp.Text)
  let ast = Parsing.Parse(tok)
  do  this.Process(ast)

[<RunAt(Side.Client)>]
member this.Process (ast:AstNode) =
  ...

[<RunAt(Side.Server)>]
member this.GenerateImg (expr:string) =
  ...
```

Figure 16. Modalities represented using .NET attributes.

The use of .NET attributes would be interesting if we could extend our work to support multiple .NET languages, but since this isn't our goal we decided to use monadic syntax which has very useful typing properties and also allows us to implement state management in the `server` monad as already mentioned. Also, the use of monads makes it syntactically easier to distinguish between the client-side and the server-side code. The only disadvantage of using monadic syntax is that it has some syntactical limitations (e.g. calling other function with the same modality has to be done using `let!` or `do!` and can't be done inside an expression).

The next consideration is whether we should allow lifting of an ordinary expression (with a non-monadic type) into a monad or not. Allowing this leads to a possibility of writing incorrect code (e.g. by using a function that is not available on the client-side), but on the other side this limitation would make it impossible to use many existing F#/.NET functions and types on the client-side without declaring a wrapper type. Bearing in mind that our goal is to allow writing the program as easily as possible, we decided to allow this and consider alternative approaches for verifying the correctness of the code (further discussed in §7.5.1).

5.4 State Management

Handling of the state in applications generally requires a big attention. In fact, the web application can be viewed as a concurrent system (with some code executing on the server and some on the client, possibly in parallel), where a state management is traditionally difficult. In addition in our web development scenario we have two environments that could possibly keep the state (client and server).

First of all, there is always a global state on the client-side, which is represented by the DOM tree and is partly managed by the web browser (e.g. it is modified when the user types a text into a textbox). The state handling on the server-side depends on the concrete application. There is usually some global state stored in the database (for the entire application) and sometimes also a global state for every user working with the application (usually called *session state* in web frameworks) however relying heavily on the per-user state is inadvisable[20].

The main reasons that influenced our design of the state management are:

- The ability to store global (per-user) state on the server is required in some situations (e.g. when storing large chunks of information generated during working with the application).
- We don't want to force storing state on the server-side (for reasons explained earlier) unless the user of our framework decides to do so.
- Since the goal is to build a system that will be easy to use for ASP.NET developers we want to choose a solution compatible with the ASP.NET programming model.

[20] There are several reasons for this – due to the stateless nature of the HTTP protocol, it is difficult to implement sessions reliably and also sessions limit either scalability (when the state is kept in memory) or efficiency (when the state is kept in database) of the server-side code.

In the presented work the state can be stored on the client-side as a field (class member) of a page or a control on the page, in addition it is possible to mark a field that should be available on the server-side when executing calls to the server. This creates a local copy of the state on the client-side and tracks changes to the state when executing code on the server, so that the changes can be performed on the client-side state after call to the server completes. The implementation is further discussed in section §6.5.

There are several alternative options, first is using the continuation passing style (where the state could be kept as an argument of a recursive function), but unfortunately there are no good ways for encoding concurrency using this model. More promising solution seems to be using some variant of the actor-based model (e.g. [32]) with message-based communication between the actors, partly extending the Links project [3] model, where messages are used for event handling. A few possible options for evolving the state-management in this work are discussed in §7.5.3.

5.5 Security

Security is an important aspect of any application and so we find it important to mention a few security implications of our project. In general in Ajax applications it is important to understand that any input from the client-side may be bogus – this is because the execution of the JavaScript code on the client-side is controlled by the client-side and the client can for example modify the JavaScript in any way he wants before executing it. This can cause many problems to an Ajax web application that for example verifies some input on the client-side and then uses the data on the server-side without verifying it again on the server-side (which is essential, because the client-side verification can be altered). It is however possible to secure data that come from the server-side, are not intended to be modified on the client-side and are sent back to the server, because the information is never modified on the client and so it can be sent encrypted using a key private to the server.

In context of our project this means following:

- For calling a server-side method from the client-side the identificator of the method is stored in an encrypted form generated on the server-side, which means that the client-side (even when altered by the user) can make calls only to methods called explicitly from the client-side code.
- The data sent to the server-side methods are not automatically verified in any way, which means that the argument to the server-side may be bogus and may not satisfy properties that were checked on the client-side.

This possibility to send a bogus data to a server-side method is indeed a problem however there are no standard ways for preventing this behavior. If the developer is aware of this problem, than the fact that calls between the server-side and the client-side are explicit makes it easy to see this in the code, so it is recommended to follow similar rules as when writing ordinary Ajax applications.

Chapter 6 Implementation

6.1 F# to JavaScript Translation

The presented translator understands with a few exceptions all F# language constructs and also allows the use of standard F# types (discriminated unions, records, tuples, lists and arrays). Implementing support for most of the functional programming constructs used in F# in JavaScript is relatively easy task, because JavaScript supports first-class functions and emulating basic discriminated unions and tuples using objects is straightforward. Additionally, no special care is needed to support list types, because F# lists are represented in terms of decimated unions. Additionally, to support sending of types 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. The asynchronous programming on the client-side is not explicitly supported by the F# to JavaScript translator and so is discussed further in a separate section (§0).

6.1.1 Functional Programming in JavaScript

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. First difficulty with JavaScript is that it 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 differently conditional expression with `unit` return type and conditional expression that returns a value. JavaScript supports the following language constructs:

```
Conditional statement:
  if (<expr>) <stmt> else <stmt>
Conditional expression:
  <expr> ? <expr> : <expr>
```

When generating code for an `if` construct that returns a value, we need to generate JavaScript conditional expression and wrap statements that can be in a body into an expression as demonstrated in the following example:

```
// F#
if (a = 1) then sideEffect(); 1 else 5

// JavaScript
(a == 1) ? (function() { sideEffect(); return 1; })() : 5
```

Second problem that we encountered is a different variable scoping in JavaScript. According to the specification, scope of any variable is the entire function where the variable was defined. This can cause problems when translating a code where one variable is reused 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 the "`i`" variable is mutable, so in the previous code, we already did one obvious workaround and copied the value of the variable to temporary variable "`x`", because using "`i`" in the lambda function created in the loop would create a reference to a mutable variable. This workaround however isn't enough, since the scope of "`x`" is the entire function body and so it exists only once and it is mutated during the execution of the loop (the created lambda functions again contain just a reference to a single mutable variable). Our translator resolves this issue by generating a new JavaScript function to produce a variable with the same scope as it would have in the F# code:

```
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 to overcome this issue is to use JavaScript `setInterval` function which executes given continuation in newly created context (after specified time), the second option is to generate a trampoline (wrap a call in a loop and throw exception with continuation when depth reaches some level). In our current implementation we don't automatically generate any of the two outlined options, mainly because we didn't find any convincing example where it would be required. For example the (infinitely running) recursive function in Figure 11 uses a `Timer.AsyncSleep`, which is internally using a `setInterval`, so it doesn't suffer from this problem and writing it without this primitive would be erroneous, because without any interruption it would block the web browser.

6.1.2 F# Core Types

The translator supports all F# types (tuples, discriminated unions, records, arrays and objects), with an exception that member augmentations[21] (for unions and records) are not supported. Object types are more complex and will be discussed later, for the other types we need to store the type of the value, because it is allowed to send values of standard F# types from client to the server (and vice versa), so the information about the type has to be maintained even on the client-side. The following table gives an overview of JavaScript representations of common F# types:

[21] Member augmentations allow adding object-like members (methods and properties) to record and discriminated union types. These members can be later accessed using the dot-notation, as members of object types.

F# type	**JavaScript representation**
tuple	Object type with signature shown in Figure 17
union	Object type with signature shown in Figure 18
record	JavaScript object
array	JavaScript array

In general, we store two additional fields for every object. These two fields are attached to every value, thanks to the fact that it is possible to dynamically append fields to any JavaScript value. These fields are used when serializing the JavaScript value and deserializing it on the server-side in order to build a F# value of the right type:

Field	**Description**
`__net_type__`	String representation of the .NET type
`__js_special__`	Determines kind of other than object type

The two types that require additional handling (tuple and union) are stored using types written in F# and translated to JavaScript. The elementary operations that can be done on the F# tuple type are creating a tuple using array of values and reading a value at specified index. Of course, in F# this is done in a type safe way, but on the JavaScript side we can use just an array of objects, because there are no static type checks.

```
type Tuple =
  new : obj[] -> Tuple
  member Get : int -> obj
```

Figure 17. Signature of the `Tuple` type, which represents an F# tuple in JavaScript

The elementary operations provided by discriminating union are creating a union value using the “tag” name and array of values, testing whether the union was created using the specified “tag” and finally reading a value at specified index (which also verifies the “tag”).

```
type DiscriminatedUnion =
  new : string * obj[] -> DiscriminatedUnion
  member IsTag : string -> bool
  member this.Get : string * int -> obj
```

Figure 18. Signature of the `DiscriminatedUnion` type, which represents an F# discriminated union value in JavaScript

6.1.3 Class-Based OOP in JavaScript

Since part of this work requires compiling (class-based) F# objects to the JavaScript, we also have to face a question how to simulate class-based OOP in JavaScript. There already exist well tested JavaScript libraries to do this, so in the presented work we use one of them, namely ASP.NET AJAX JavaScript framework for class-based OOP simulation [7].

To give just a quick overview of the OOP framework, the following example demonstrates the type from the F# introduction (§2.1.4) written in JavaScript using the ASP.NET AJAX library:

```
MyCell = function (n) {
  this.data = n + 1;
};

MyCell.prototype = {
  get_Data : function () {
    return this.data;
  },
  set_Data : function (v) {
    this.data = v;
  },
  Print : function (v) {
    window.alert("Data: " + v);
  }
};
MyCell.registerClass("MyCell");

MyCell.FromInt = function(n) {
  return new MyCell(n);
}
```

Our translator follows this pattern, so the objects generated by our translator can be used from other client-side code written directly in JavaScript, though we make two important changes.

First we want to allow classes with more than one constructor, which is possible in F# thanks to the overload resolution). To allow this, our translator moves constructor code into function with special name and all calls that create instance of an object are modified accordingly to explicitly call the correct constructor. For types that can be used from both server-side and client-side, there are also a few extensions to allow serialization of such types. These extensions are discussed below in §6.5.

6.2 Rich Client-Side Environment

6.2.1 Internal Mappings

As already demonstrated in §4.1, internal mappings are used for accessing native JavaScript functions from the code written in F#. The mapping is implemented using the `Mapping` attribute, which defines the name of the native function and also its scope and a way in which it is called. The first parameter (value of `MappingType` enumeration) specifies target language construct in JavaScript:

Type	**Use in F#**	**JavaScript code**
`Method`	`any`	`[<inst>.]Foo(<args>)`
`Property`	`get`	`[<inst>.]get_Foo(<args>)`
`Property`	`set`	`[<inst>.]set_Foo(<args>)`
`Field`	`get`	`[<inst>.]Foo`
`Field`	`set`	`[<inst>.]Foo = <arg>`
`Object`	`any`	`new Foo(<args>)`
`Inline`	`empty <args>`	`Foo`
`Inline`	`with <args>`	`Foo(<args>)`

The “use in F#” column specifies what F# language constructs are compatible with the specified type of mapping – this for example allows declaring mapping to JavaScript properties or fields only as an F# property, because otherwise it wouldn’t be clear when it is used for reading and when for writing a value. In cases where the instance is optional, it is possible to specify whether the target JavaScript construct expects the instance or not. This can be done using the second parameter (value of `MappingScope`) which has the following values:

Type	**JavaScript code**	**Arguments**
`Global`	**`Foo`**	*`<inst>::<args>`*
`Member`	*`<inst>`*`.`**`Foo`**	*`<args>`*

When mapping an instance call from F# to a JavaScript call with global scope (`MappingScope.Global`), the instance argument is given as a first argument to a call followed by remaining arguments, conversely when mapping F# static call to a JavaScript instance call, the first argument is used as an instance.

The following code demonstrates a way for accessing JavaScript `alert` function, which is a member of the global `window` object via static method call in F#:

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

Figure 19. Mapping for the `window.alert` function.

Since the mapping to an alert function requires an instance (its scope is set to `MappingScope.Global`), the translator finds a mapping for the `Window` type, which is a global field (field accessed without specifying an instance) and it produces the expected JavaScript code (`window.alert(arg)`).

6.2.2 External Types and Modules

The external mappings can be defined for classes 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 can be used in client-side, which means that they can consist only of client-side F# code or internal mappings. The possibility to combine both types of mappings in a single type is very powerful as it is often possible to map some functionality to existing JavaScript functions, but some of the more advanced functions have to be reimplemented. The combination is demonstrated in Figure 20.

```
[<ClientSide; ExternalType(type System.Int32)>]
type Int32 =
  [<Mapping("tostr", MappingScope.Global, MappingType.Method)>]
  override x.ToString() : string = (raise ClientSideScript)

  [<ReflectedDefinition>]
  static member Parse(s:string) =
    // ...
```

Figure 20. A demonstration of using combination of internal and external mappings.

6.2.3 Client-side DOM framework

An important part of the client-side environment is a library for manipulating with the DOM (document object model), which is a standard way that browsers expose for working with the displayed HTML page. It is not our goal to document the entire DOM framework in this text, however we'd like to demonstrate how it is designed using a few examples. The following code is a subset of the implementation of the calendar control that was demonstrated in the Lecture Organizer case-study (§4.3):

```
member this.GetDayElement(i) =
  client
   { match this.GetChild("day"+i.ToString()) with
     | Some el -> return el
     | _ -> return failwith "Element not found!" }

member this.AttachHandlers () =
  client
   { for i in [1, this.monthInfo.TotalDays] do
       let! el = this.GetDayElement(i);
       do   el.ClientClick.AddClient(fun (_, e) ->
                 this.DayClicked(i, e)) }

member this.DayClicked(i:int, e:CancelEventArgs) =
  client
   { // ...
     let! oel = this.GetDayElement(d)
     let! nel = this.GetDayElement(i)
     if (...) then oel.RemoveCssClass("sel")
     nel.AddCssClass("sel")
```

Most of the processing works with an abstract type that represents generic HTML element – the control itself is a subclass of this type, because every control has to be contained in an element. The children of the element can be accessed using the `GetChild` method, which returns an element using the F# option type (this demonstrates our goal to make the framework friendlier to a F# developer). Some of the members of this abstract element type are demonstrated in the following table:

Member	**Type**	**Description**
ClientID	string	Returns the ID of the HTML element
Visible	bool	Controls visibility of the element
InnerHtml	string	The HTML code inside the element
AddCssClass	string -> unit	Adds specified CSS class
RemoveCssClass	string -> unit	Removes specified CSS class
ClientClick	ClientEvent	Raised when the element is clicked
ClientMouseUp	ClientEvent	Raised when a button is pressed
ClientMouseDown	ClientEvent	Raised when a button is released

The design of the client-side GUI framework is indeed mostly following the DOM implementation in the web browsers however we adjusted it to better fit with the F# programming style. The fact that the framework is a wrapper around browser implementation also allows us to hide certain incompatibilities between implementation in web browsers.

6.3 Client-Server Integration

The integration between the client and the server-side is probably the most interesting part of the presented work. As mentioned earlier, the modality of a code can be either expressed using a monad (for single functions) or using a .NET attribute for entire class or F# module. The following attributes can be used in the second case:

Attribute	**Modality**
`MixedSide`	Vary for all members, depends on the monadic type
`NeutralSide`	Available on both sides (*neutral*)
`ClientSide`	Available only on the client-side (*client-side)*
(none)	Available only on the server-side (*server-side*)

If the used attribute is `MixedSide` than the modality is different for every member of the type and is determined using the type of the member, which is indeed specified by the monad used when writing the code. Aside from client, server and neutral code, the monadic type can also specify asynchronous client-side code:

Builder	**Monadic Type**	**Modality**
neutral	NeutralMonad<’a>	*neutral* (client and server)
client	ClientMonad<'a>	*client-side*
server	ServerMonad<'a>	*server-side*
client_async	ClientAsyncMonad<'a>	*asynchronous client-side*

Indeed, we could use only the monadic syntax to distinguish between different kinds of the code, but since we want to allow using certain standard F# and .NET libraries, we have to provide a way for accessing types that don’t use our monadic modality annotations. Additionally, when an entire module or a class has the same modality the monadic syntax feels too verbose and adding annotation using .NET attribute with larger scope allows us to write more readable code.

This way of separating a code that is executed in different environments is also extensible and it is easy to imagine using other monadic builders for writing code with other modalities. We discuss several possible extensions later in (§7.5.1). The different modalities mentioned in this overview are discussed in detail below.

6.3.1 Monadic Builders for Modalities

Before mentioning details of each monadic builder, we need to explain one relaxation of standard monadic rules that we use. In the overview there are a few modalities where it is natural to allow calls from one to another via the bind operators (`let!` and `do!`). The vital example is the neutral code, which can be by design called from both client and server-side code, meaning that we want to allow something like this:

```
// neutral function that adds two numbers
let add(a,b) =
  neutral
    { return a + b }

// using neutral function from the client-side
let clientTest() =
  client
    { let! res = add(1, 2)
      return res }

// using neutral function from the server-side
let serverTest() =
  server
    { let! res = add(1, 2)
      return res }
```

Using the standard monadic implementation, this code would be obviously incorrect, because the types of the `add` function wouldn't be compatible with either of the bind operations:

```
add         : int -> int -> NeutralM<int>
client.Bind : ClientM<'a> -> ('a -> ClientM<'b>) -> ClientM<'b>
server.Bind : ServerM<'a> -> ('a -> ServerM<'b>) -> ServerM<'b>
```

Hence the question is, if there is any reasonable relaxation of these types that we could use to allow writing a code using this style. Note that F# doesn't restrict the type of the monadic operators in any way, though it is recommended to use the standard typing scheme.

In this work we use subtyping to deal with this problem. Instead of requiring exactly the same monadic type in the bind operator, we declared a supertype[22] for every modality that can have multiple subtypes, all of them useable as an argument for the bind operator. This means that the types of the functions relevant to the previous example are actually following[23]:

```
add         : int -> int -> NeutralM<int>
client.Bind : #IClient<'a> * ('a -> #IClient<'b>) -> ClientM<'b>
server.Bind : #IServer<'a> * ('a -> #IServer<'b>) -> ServerM<'b>
```

Where the following relations hold for the `NeutralM<'a>`, allowing us to use neutral code as an argument for bind operators of both client and server-side code[24]:

```
NeutralM<'a> <: IClient<'a>
NeutralM<'a> <: IServer<'a>
```

And of course the following holds as well, to enable calling client-side code form other client-side code (and the same for server-side):

```
ClientM<'a> <: IClient<'a>
ServerM<'a> <: IServer<'a>
```

22 Because some types may need to have more than one supertype, we use an F# interface, which allows us to use multiple interface inheritance.
23 We're using an F# notation where #T means any subtype of T.
24 We're using notation from [34], where „<:" is a subtype relation.

It is important to note that using this relaxation some of the monadic laws formalized in [35] do not hold for our implementation. We are aware of this fact and we find it important to describe (at least informally) how our approach affects these three laws. The laws that don't hold in our implementation are the *left unit law* and the *right unit* law, while the third (*associative*) law holds. The rules are expressed using the following equation (adopted from [35]):

$$\begin{array}{ll} unit\ a * \lambda b.n\ = n[a/b] & \text{(left unit)} \\ m * \lambda a.unit\ a\ = m & \text{(right unit)} \\ m * (\lambda a.n * \lambda a.o) = (m * \lambda a.n) * \lambda a.o & \text{(associative)} \end{array}$$

In the following text we assume that the type of the bind and return operations are following (this is the scheme used for all monadic builders in our implementation):

```
m.Bind   (*)    : #IM<'a> -> ('a -> #IM<'b>) -> M<'b>
m.Return (unit) : 'a -> M<'a>
```

And the following subtyping relation exists:

```
M<'a> <: IM<'a>
```

The first law requires that computing a value of *a*, binding it to *b* (using the monadic bind) and computing *n* produces the same result as executing *n* with *a* substituted for *b*. The reason why this does not hold in our implementation is that the type of *n* may be any subtype of `IM<'a>`, but not necessarily `M<'a>`, while the result of the monadic operation on the left hand side will be `M<'a>` in all cases and so the types may differ.

The second law requires that computing *m*, binding the result to *a* (using monadic bind) and returning the result (using monadic return) produces the same result as executing *m*. It doesn't hold for similar reason as the first law – in our implementation, the type of *m* is any subtype of `IM<'a>`, while the value produced by the expression on the left hand side is of type `M<'a>` in all cases.

Finally, the third law holds in our variation, assuming that it holds in the implementation of the monad. First, since the types of all *m*, *n* and *o* values are all subtypes of the `IM<'a>` type and thanks to the subtyping relation between `M<'a>` and `IM<'a>`, it is sound to write the whole equation (i.e. it is possible to sequence the bind operations). Second, the result type is on both sides the result type of the bind operation, which means that the type of both expressions will have a type `M<'a>`.

In our case this means that using a monadic bind operator may restrict the modality of the value (e.g. when we use bind of the client monadic builder on a value of a *neutral* code type, we will get a *client* code type), which is a reasonable and expected behavior.

6.3.2 Concrete Modalities in Detail

Server-Side Code

Let us start with the server-side code. When the type is written using monadic syntax, it is introduced by the `server` monadic builder, whose bind operation accepts values of any subtype of the abstract type `IServer<'a>` and produces values of concrete type `ServerM<'a>` meaning that the type of the bind operation is following:

```
server.Bind : #IServer<'a> -> ('a -> #IServer<'b>) -> ServerM<'b>
```

The subtype of the abstract type `IServer<'a>` provides a way for reading a value of type `'a`, meaning that the server-side code is executed while the monadic operators are called, which is the simplest possible implementation of building and executing monadic computations. The `ServerM<'a>` is just a trivial implementation of such abstract type, meaning that it contains a value of type `'a`, though in fact the `IServer<'a>` type has one more aspect to enable state management, which is described below.

Server side code can be called only by other server-side code and the other way round, server-side code can call other server-side computations or a neutral code, which is expressed by the following subtyping relation in the monadic syntax:

```
ServerM<'a>  <: IServer<'a>
NeutralM<'a> <: IServer<'a>
```

The subtype relation in this case has a meaning that a computation with the modality of the subtype (the server-side or the neutral computation) can be used in the computation with a modality of the supertype (server-side code in this case). The correctness of the use of monadic typing as well as other rules for the server-side code that can't be verified by a type checker are discussed later in a section §6.3.3.

Client-Side Code

The computations that involve client-side code have similar properties to the server-side code mentioned in the previous section. Client-side monadic computation is introduced by the `server` monadic builder, whose bind operation has the following type:

```
client.Bind : #IClient<'a> -> ('a -> #IClient<'b>) -> ClientM<'b>
```

Since the client-side computations are never executed on the server-side (i.e. in the native compiled code) neither the abstract nor the concrete types have any members and only one (dummy) value of the concrete type exists. The client-side code is always executed in the browser as a JavaScript generated by the translator, which also means that every client-side member has to be marked using the `ReflectedDefiniton` attribute to allow translating the F# code to JavaScript using meta-programming (as described in §2.1.5).

The subtyping relations involving the client-side computation type are however crucial for ensuring that the computations are composed in a correct way:

```
ClientM<'a>  <: IClient<'a>
ClientM<'a>  <: IClientAsync<'a>
NeutralM<'a> <: IClient<'a>
```

These relations say that client-side code can be called by other client-side code or by asynchronous client-side code (see below) and conversely client-side code can also call neutral code.

Neutral Code

Neutral code is used to represent computations that can be executed on both client-side and server-side. "Neutral" may be a bit misleading name from the implementation point of view, because the neutral code has to support both execution on the client-side, requiring the translator to understand the types used when writing neutral code, and on the server-side, which requires inheriting the `IServer<'a>` abstract type. The bind operation of the monadic builder for neutral code (introduced by the `neutral` value) has following type:

```
neutral.Bind : #INeutral<'a> -> ('a -> #INeutral<'b>) -> NeutralM<'b>
```

Subtyping relations involving the neutral code types are following:

```
NeutralM<'a> <: INeutral<'a>
NeutralM<'a> <: IClient<'a>
NeutralM<'a> <: IServer<'a>
NeutralM<'a> <: IClientAsync<'a>
```

This means that the neutral code can call only neutral code, but can be called by all other kinds of computations (neutral, client-side, server-side and also asynchronous client-side). From the implementation point of view, the neutral code limits extensibility of our solution because it needs to support execution in all existing environments. On the other side there is one very appealing reason for having such concept of neutral code – it is often possible to give an implementation of some basic operation in both client-side and server-side code and when this is possible it would be appealing to allow building a neutral computation from these two environment-specific computations to allow using the composed computation from operations written as a neutral code.

This composition of computations is indeed supported by our implementation and can be done using the primitive function `buildNeutral` which has the following type:

```
buildNeutral : #IClient<'a> * #IServer<'a> -> NeutralM<'a>
```

The following example demonstrates building a neutral computation for changing a size of a popup window and using it from a neutral computation. When changing the size on the server-side we just update a local field that represent the size (and will be used for rendering the window), but on the client side we also have to update the window if it is displayed:

```
[<ReflectedDefinition>]
member this.SetWindowSize(sz) =
  buildNeutral
   (server { do this.size <- sz },
    client { do this.size <- sz
             if (this.visible) then
               this.UpdateWindow() })
```

Similar constructs are found very often in basic controls in the F# Web Toolkit – for example setting a text of a label or any similar operation can be performed on both client-side and the server-side, but using a different implementation.

Asynchronous Client-Side Code

Asynchronous client-side computations are introduced by the `client_async` monadic builder. Similarly as in previous cases, the abstract monadic type (`IClientAsync<'a>`) and a concrete monadic type (`ClientAsyncM<'a>`) exist to allow implicit calls between different types of computations subtyping relations that allow this are following:

```
ClientAsyncM<'a> <: IClientAsync<'a>
ClientM<'a>      <: IClientAsync<'a>
NeutralM<'a>     <: IClientAsync<'a>
```

This means that the asynchronous client-side code can be called by any other asynchronous client-side computation and can also call neutral computations and ordinary client-side computations.

Even though the code runs on the client-side (as a JavaScript produced by the translator) the `ClientAsyncM<'a>` type is not just a dummy type (as in case of ordinary client-side code), because the JavaScript translator produces similar code as the F# compiler by desugaring the monadic syntax (as described in §2.1.6) and the `ClientAsyncM<'a>` type is translated to JavaScript as well and is used to keep a value representing the monadic computation (in JavaScript). The use of the same type for representing the value in the F# code and in the produced JavaScript is crucial to allow writing primitive asynchronous computations as described later.

The builder for this monadic type, which exists in JavaScript, implements essentially the continuation monad, where the computations have a type usual in continuation monads:

```
type ClientAsyncM<'a> = | ClientAsync of (('a -> unit) -> unit)
```

This can be read as "a function that will generate an `'a` value sooner or later and it will call the continuation (given as an argument) when the value is available".

As already mentioned, thanks to the fact that the value is written as ordinary F# type, it is possible to write custom primitive[25] asynchronous operations. To demonstrate this we show source code of the asynchronous sleep function which stops the execution for specified amount of time (and which we used in earlier examples) in Figure 21. It is obvious that the same pattern can be used to wrap waiting for any native JavaScript event (e.g. waiting for a mouse click) in an asynchronous operation.

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

[25] By "primitive" we mean an operation that is not composed from other asynchronous values, but uses some native JavaScript functionality for implementing the asynchronous behavior.

```
          t.Stop();
          cont() )
        t.Start(); )
```

Figure 21. F# source code for the `SleepAsync` function.

The generated JavaScript code is demonstrated in Figure 22. The F# code isn't significantly shorter, but was much easier to write thanks to the static typing guarantees.

```
function SleepAsync(ms) {
  return CreateObject(new DiscriminatedUnion(),
    [ "ClientAsync",
      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 22. JavaScript generated for the SleepAsync function.

6.3.3 Separation Correctness

It is of course important to discuss a set of rules that have to hold in the code written using the presented way in order to make sure that the separation between different kinds of code modalities is correct, meaning that the code executed in one environment can't make calls to a code intended to run in the other execution environment with an explicitly allowed exceptions (for example using the `serverExecute` function).

Our ultimate goal is to be able to verify all these rules at compile-time, but in the current implementation, the compiler verifies only rules encoded in the monadic typing. We first introduce rules that use monadic typing and the rules that are not verified automatically and have to be kept in mind when writing the code are presented later.

Modalities using Monadic Typing
First, we informally review all possible combinations of calls that can occur in code written using three basic modalities (*client*, *server* and *asynchronous client*) and the *neutral* modality is discussed below. In the discussion we use functions with the following signatures:

```
val serverF  : unit -> ServerM<unit>
val clientF  : unit -> ClientM<unit>
val clasyncF : unit -> ClientAsyncM<unit>
```

Since we introduced a subtyping relation between modalities earlier in the text, we have to consider subtyping when discussing what types of code can invoke a code written using the monadic modality type. In the following text we will use symbol "<:" to represent a subtyping relation and a symbol "□" to represent that there is no relation between two types. First, we consider options when client-side code invokes one of the functions declared earlier:

```
client { do! clientF()  }    // OK   (C<unit> <: IC<unit>)
client { do! serverF()  }    // FAIL (S<unit> □  IC<unit>)
client { do! clasyncF() }    // FAIL (A<unit> □  IC<unit>)
```

From these cases, only a call between the same modalities is allowed, which means that a client-side code can other call client-side code (this is clearly correct). Call to an asynchronous client-side code is allowed only explicitly using `asyncExecute` function with the following signature:

```
asyncExecute : IClientAsync<unit> : ClientM<unit>
```

It is important to note that the return type of the code executed using this function has to be `unit` (i.e. it has no return value), because the semantics of the function is that the code is spawned on a different "thread". The next section discusses calls from the server-side code:

```
server { do! clientF()  }    // FAIL (C<unit> □  IS<unit>)
server { do! serverF()  }    // OK   (S<unit> <: IS<unit>)
server { do! clasyncF() }    // FAIL (S<unit> □  IS<unit>)
```

Clearly, a server-side code can call only other server-side code. Calls to a client-side are not possible (not even explicitly), because a server-side code is not executed in a continuation-passing style and so calling a client-side code is by design impossible. Finally, the calls from the asynchronous client-side code are following:

```
client_async { do! clientF()  }  // OK   (C<unit> <: IA<unit>)
client_async { do! serverF()  }  // FAIL (S<unit> □  IA<unit>)
client_async { do! clasyncF() }  // OK   (A<unit> <: IA<unit>)
```

As we can see, the asynchronous client-side code can call other asynchronous client-side code. This is done using the monadic `Bind` operation on the monadic type representing an asynchronous client-side code (a continuation monad). The call to an ordinary client-side code is allowed implicitly using a subtyping, though this has to be handled explicitly in the F# to JavaScript translator as the call is done using the monadic `Let` operation (which represents a binding of a non-monadic value). Finally, the call to a server-side code is allowed only explicitly using the `serverExecute` function which executes a call asynchronously and then calls the continuation with a result returned from the server-side. The signature of this function is following:

```
serverExecute : IServer<'a> -> ClientM<'a>.
```

As discussed earlier, the code with the *neutral* modality is a subtype of all three modalities, namely server-side code (`IServer<'a>`), client-side code (`IClient<'a>`) and client-side asynchronous code (`IClientAsync<'a>`). The subtyping relation implies that it should be possible to use this kind of code in a place where any other type of code is expected. For the server-side code, this is possible because the `NeutralM<'a>` type, which represents a code with this modality implements all functionality required by the server-side code (by implementing the `IServer<'a>` interface). On the client-side, the code with `neutral` modality is translated to an ordinary client-side code (equivalent to a code written with the client-side modality only), so the calls from client-side code to the neutral code are translated in an exactly same way as calls to other client-side code blocks. Finally, when calling a neutral code from the asynchronous client-side code, it is treated in a same way as ordinary client-side code and it is called using the monadic `Let` operator.

F# does not support co-variance or contra-variance, neither when working with generic types, nor for method overriding, so it is not necessary to discuss a cases where variance would be involved, but the relations between modalities of the code have the same properties as a subtyping relation, so the presence of variance would not affect correctness of our approach.

Lastly, it is important to note that F# doesn't automatically perform upcasts when calling a function and so a function can be declared with a signature that allows arguments of type `T` or arguments of type `T` and any subtypes of type `T`, which is written as `#T` as demonstrated by the following two functions:

```
val map1 : list<'a> -> ('a -> #IClientAsync<'b>) -> list<'b>
val map2 : list<'a> -> ('a -> IClientAsync<'b>) -> list<'b>
```

The following example demonstrates how these functions can be used:

```
map1 [ 1 .. 10 ] client   // Correct
map2 [ 1 .. 10 ] client   // Not correct – incompatible types

// Correct – the argument has the corresponding type
map2 [ 1 .. 10 ] (fun n -> client { return! (clientF n) })
```

Additional Modality Rules
Other rules that are not tracked by the type system are introduced when working with types or modules where the modality is defined using an annotation (either using the `ClientSide` attribute for client-side code or using the `NeutralSide` attribute for a code with neutral modality). As mentioned earlier, it would be appealing if it were possible to verify these rules during compile-time as well a possible solution, which would be implementing an extensible verification for the F# compiler is mentioned in §7.5.1.

The types that contain internal mappings described in §6.2.1 have to be marked using the `ClientSide` attribtue, so the verification of internal mappings doesn't require any additional handling. Further, any type for which an external mapping is defined (meaning that a client-side implementation of a type that exists on the server-side) is treated as a type marked using the `NeutralSide` attribute. Lastly, any type or module not marked using neither of these two attributes is treated as a server-side code.

The attributes define a code modality for the entire type which means that any code present in the type/module is treated accordingly to the attribute which specifies the modality. The rules for a calls across different modalities in these types are the same as these expressed using monadic modalities, which means that neutral code can call only neutral code, server-side code can call only server-side or a neutral code, client-side code can call only client-side or neutral code.

6.4 Serialization and Mixed Types

6.4.1 Mixed Types

Mixed type is a term that we use when talking about type that can exist on both client-side and the server-side. Mixed types serve as a container for the code with mixed modalities written using the monadic syntax and so they are probably the most important part of the work from the object-oriented point of view.

The most common cases where mixed types are used in our project are objects representing pages and controls and indeed, we already used them in a few examples. The only unusual aspect of mixed types that represent GUI elements is that construction of these is controlled by the GUI framework, which builds them according to the declarative markup (in the `aspx` file). Controls are discussed in more detail below in section §6.5.

In some sense mixed types combine the client-side part of the type and the server-side part of the type, but the key benefit (when comparing to working with two separate types) is that some parts of the type can be neutral, and so are available to both sides. A parts of the type that can be neutral are members that can be executed in both environments (written using `neutral` monadic type) and fields (marked using a special attribute) that are automatically serialized when transferring control flow from one environment to another. All possible types of members that can be used in a mixed type are demonstrated in Figure 23.

```
[<MixedSide>]
type MixedDemo =
  // Field available only on the client-side
  [<ClientField>]
  val clientStr : string
  // Field available only on the server-side
  val serverStr : string
  // Field shared by both sides
  [<NeutralField>]
  val num : int

  // Client-side member
  member x.ClientMethod() =
    client { ... }
  // Server-side member
  member x.ServerMethod() =
    server { ... }
  // Member available on both sides
  member x.NeutralMethod() =
    neutral { ... }
```

Figure 23. Possible types of members that can be used in a mixed type.

6.4.2 Mixed Type Semantics

Formal definition of semantics of mixed types isn't a goal of our work, but we'd like to present what rules we adopted in our implementation as well as several interesting problems related to splitting a single type between two execution environments. The two problems that we discuss further are object-oriented inheritance and construction of these types.

Inheritance

The first concern when writing mixed types is what type can a mixed type inherit from? The problem can be viewed as if we had two different types, one for the server-side and the one for the client-side, where the mixed part is duplicated in both of the types. The most common case of inheritance that we clearly have to allow is inheriting a mixed type from another mixed type, in which case the client-side and the server-side part of the subtype inherit from the client-side or the server-side part of the supertype respectively. This is the option used most often by users of our framework and so we will discuss it in a larger detail and mention other possible inheritance relations later.

In our implementation the server-side code is executed natively using the .NET runtime and the client-side code is executed as a JavaScript code (translated using the OOP emulation framework for JavaScript), which means that the inheritance relation exists in a different notion on both sides. Inherited mixed type is demonstrated at the following example (including two overridden members, one on the server-side and one on the client-side):

```
[<MixedSide>]
type InheritedMixed =
  inherit BaseMixed
  override x.ServerFoo() =
    server { return 1 }
  override x.ClientFoo() =
    client { return 1 }
```

The server-side code is compiled into a .NET executable, where the inheritance relation exists (as in any ordinary F# code). In the client-side code the JavaScript class that is generated from the type inherits from the JavaScript class generated from the base type.

The other option that we find reasonable is to allow the inheritance to exist only in one of the environments, for example having a mixed type inherited from the server-side type or from a client-side type. In these cases the inheritance relation exists only in one environment and in the other a new type is introduced. Finally, it is also reasonable to consider a case where the type inherits from two different types, one being a server-side type and the other client-side type. In this case the two base types are combined, but it doesn't lead to common problems with multiple inheritance since there can't be any relation between the two base types. These additional options are useful mainly for extending an existing code and so we use the last option internally in our framework for building a base types for user interface elements (such as pages and controls), however these options have to be used very carefully and our implementation focused mostly on a cases needed by our framework.

Control Flow
When discussing how the mixed types can be created and how the control flow is managed, it is important to realize that there is an asymmetry in how the objects can be used – this is in particular because the client-side can make calls to the server-side (using the `serverExecute` function), but not vice versa.

Our implementation allows creating an object on both server-side and the client-side. When the object is created on the server-side it can't however make calls to a client-side and so the entire lifetime of the object is limited to a server-side. On the other side when the object is created on the client-side it can execute some code on the client-side and then transfer the control flow to a server side. As mentioned earlier, the object may contain a mixed-side fields that are automatically serialized and sent between the both environments so the object created on the client-side can make a call to the server, the server-side code can update a side-neutral field and when the execution of the server-side code finishes, the changes made to a neutral-side fields are propagated back to the client-side. The implementation of this field-update tracking mechanism is discussed in the next section.

6.4.3 State Management for the Server Calls

When talking about the `ServerM<'a>` type earlier in §6.3.2 we mentioned just its typing properties that are important for the correctness of the client-server integration, but we didn't describe in a detail how it supports the scenario presented in a case study §4.3, where the `ServerM<'a>` is used to collect "state updates" executed on the server during an asynchronous call from the client, so that the changes can be performed on the client-side after the call finishes. Using the terminology from the previous section, the members of the `ServerM<'a>` type can be used only in mixed types and the "state updates" are updates of the value of neutral-side field.

The `ServerM<'a>` is an implementation of the abstract type `IServer<'a>` which has the following structure:

```
// Stores a single state update
type StateUpdate =
  { TargetControl : obj;
    SetProperty   : bool;
    PropertyName  : string;
    Value         : obj;    }

// Represents a value and list of state updates
type IServer<'a> =
  abstract Value : unit -> 'a
  abstract StateUpdates : StateUpdate list
```

As we can see, the server-side computation contains a value (which is the result returned by the computation) and a list of state updates. State update is represented by a data type that contains a reference to the target control, a flag indicating whether we updated a field (in this case we will just set a value of the field in JavaScript) or a property (which means that some associated JavaScript code may be executed when setting it). It also contains the name of the property or a field to set and of course a value to be set. The monadic builder (`server`) that produces values of this type is easy to implement – the `Return` operation just calculates the result and produces a monadic value with empty list of side-effects and the `Bind` operation in addition concatenates the two lists as demonstrated in Figure 24.

```
// Monadic bind operation for the server monadic type
let bind g (a:#IServer<'a>) (f:'a -> #IServer<'b>) =
  // Call the function
  let b = (f a.Value)
  // Concatenate the state updates
  let stateUpdates = a.StateUpdates @ b.StateUpdates
  // Build a new value of type Server<'b>
  makeServer (b.Value) (stateUpdates)
```

Figure 24. Implementation of the bind operation from the ServerM<'a> monad.

In general it would be possible to allow tracking calls to any client-side code by extending the structure to allow representation of a method call as well, but our intention when introducing this feature is to allow state management and not delaying execution of some client-side. There are two main reasons for this intentional limitation, first is that it could easily introduce bugs in the code, because the fact that the computation is delayed (which is rather hidden in the code) changes the semantics of the code and the second is that allowing a method call would lead to an obvious question: "How can we use a method that returns a value?", which is indeed impossible by design.

The calls to the server are possible only using the `serverExecute` function, which has a type that allows using calling a function returning `ServerM<'a>` from a code that accepts `ClientM<'a>`. The function isn't actually implemented anywhere, because it is treated specially in the F# to JavaScript translator, which recognizes the following pattern[26]:

```
client_async
  { let! <res> = serverExecute (<serverMethod>(<arguments>))
    ... }
```

The call is (just by syntactic transformation) translated to the F# code that uses functionality available on the client-side and so it is possible to translate it to JavaScript using our general purpose translator:

```
AsyncMonad.Bind
  (Async.HttpCall
    (new LibXmlHttpRequest("POST", "<encodedServerMethod>"),
    <instance>, [<arguments>]),
  fun <res> =>
    ... )
```

In this code snippet, the `AsyncMonad` type is a client-side implementation of the continuation monad which was discussed earlier (in §6.3.2). `LibXmlHttpRequest` is a client-side type that wraps functionality of the native `XmlHttpRequest` object

[26] Since the translator recognizes calls as a syntactical pattern, this brings some restrictions on a way the `serverExecute` function can be used, because server-side functions can't be treated as a first-class values, meaning that it is not possible to create a function that would take server-side function as an argument and execute it using `serverExecute`. We'd like to remove this limitation in the future.

which is used in Ajax applications for performing asynchronous calls to the server, *<encodedServerMethod>* is an encrypted string that encodes the method to be called, *<instance>* is an optional value that represents the object on which the method was invoked if the method is an instance method, or `None` for static methods.

The `HttpCall` function is similarly to the `SleepAsync` function in Figure 21 implemented in F# as a primitive asynchronous function that wraps the use of native JavaScript events. The implementation is shown in Figure 25 and uses several other objects (`ClientCore` contains wrappers for dynamic JavaScript functionality, `JsonSerializer` and `JsonDeserializer` will be discussed later).

```
[<ReflectedDefinition>]
static member HttpCall<'a> (req, inst, args) =
  ClientAsync (fun (cont:'a -> unit) ->
    req.Received.Add(fun (sender, e) ->

      // Deserialuze the response
      let ont = ClientCore.Eval(req.ResponseText)
      let ods = JsonDeserializer.DeserializeServerResponse(ont)
      let val = ClientCore.Cast<'a>(ods)

      // Call the continutaion
      (cont val) )

    // Serialize arguments and call the server-side
    let snd = JsonSerializer.SerializeCallArguments(inst, args)
    req.Send(snd); )
```

Figure 25. The implementation of the asynchronous client-side `HttpCall` function.

The call to the `serverExecute` function is the only place in the code where the automatic serialization takes place. As can be seen in Figure 25, the instance on which the call is performed is given as an argument to a function `SerializeCallArguments`, which serializes all the neutral-side fields that have to be copied to the server-side. After the call is completed, the function `DeserializeServerResponse` updates the fields modified during the call to the server-side.

The fact that the call is asynchronous introduces a concurrency in the code, because the client-side code may change a value sent to the server. In a context of web applications this concurrency usually requires different solution than traditionally, so we leave handling of this problem to a developer. Also in a web application it is often reasonable to ignore the problem, for example when the user invokes a call that loads the next page of a displayed data and then changes a view on the current page it is expected behavior that the new page will load even though something was changed in a meantime[27].

[27] A possible improvement would be to implement optimistic concurrency control and add a validation that a value didn't change during a call to a server-side code, however this would require an additional effort from the developers who use the system, because they would have to specify what to do if the validation fails.

6.4.4 Implementation of the Serialization

The serialization uses JSON format to encode the data, however it is not possible to encode all objects just using the key-value format which is sufficient for representing data in JavaScript, because the data that are serialized in our implementation are classes and we need to deserialize them as a class (with associated methods, internal information representing the .NET type of the class, etc.).

It is also important to note that the serialization has to be implemented on both server-side (when sending a value from server to a client) and client-side (when sending a value from client to a server) and similarly for the deserialization. Even though in our case all 4 operations are implemented in F# it would be difficult to share part of the implementation, primarily because the reflection mechanism that is essential for the implementation is different on both sides – on the server-side the .NET reflection is used and on the client-side this is done using dynamic features of the JavaScript language and runtime (though these are mapped to a set of simple F# functions).

Our serialization mechanism supports are standard F# types (that can be sent between the client and the server as needed) as well as mixed types (built using F# object oriented programming) discussed in this section. The patterns used for encoding the types are demonstrated in Figure 26.

```
Array
  { "__js_special__": "array", "net_type": "<encoded type>",
    "members": [ <elements> ] }

Record
  { "__js_special__": "record", "net_type": "<encoded type>",
    "members": { <key-value dictionary> } }

Discrimated Union
  { "__js_special__": "union", "net_type": "<encoded type>",
    "tag": "<tag name>", "args": <args array> }

Tuple
  { "__js_special__": "tuple", "net_type": "<encoded type>",
    "args": <args array> }

Mixed Type
  { "__js_class__": "<encoded type>", "typeargs": <tyargss arr>,
    "properties": { <key-value dictionary> } }
```

Figure 26. Encoding of serializable F# types in the extended JSON format.

Note that for a generic mixed types we store a type of the type arguments separately (in an array) and not as a part of the encoded .NET type of the class. This is needed in a JavaScript code that uses the type of the type argument when creating certain types inside the generated code. For example when a type `List<T>` initializes an array of type `T[]` it needs to know the encoded .NET type representation of `T`, so the array can be later correctly serialized and sent to the server-side.

6.5 Composable Components

As mentioned when introducing mixed types, the most common use of mixed types in our work is for representing user interface elements, specifically types representing pages and controls, which we together call components. Components inherit all features of mixed types with the only difference that the construction of components on the client-side is done automatically, using the component structure produced when initializing the page on the server-side, usually using the declarative markup.

6.5.1 Building a Page using Components

Since our project is built using ASP.NET, we can easily use the declarative way for building a page from components provided by ASP.NET. In ASP.NET it is possible to define the markup in a file with `aspx` extension and the interaction logic of the page in the file with `fs` extension (when using the F# language). A sample page that contains a textbox control (for entering a text) and an element control (for displaying response) which "pings" the server-side when the text in the textbox changes is demonstrated in Figure 28 (markup) and in Figure 27 (page interaction logic).

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

  val mutable txtInput : TextBox
  val mutable ctlOutput : Element

  member this.Client_Load (sender, e) =
    client
     { do this.txtInput.ClientKeyUp.AddClient
            (this.UpdatePing) }
  static member Ping(s) =
    server
     { return "Response to '" + s + "'." }

  member this.UpdatePing (sender, e) =
    client
     { let str = this.txtInput.Text
       do! asyncExecute
            (client_async
              { let! sugs = serverExecute(Suggest.Load(str))
                do!  this.DisplayResponse(sugs) }) }
```

Figure 27. Suggest.aspx.fs file with the interaction logic for a sample page.

The page interaction logic contains one additional extension that is not inherited from ASP.NET – this is the `Client_Load` method, which is a special method that will be invoked when the page is loaded on the client-side and can be used for initialization of the page. In the example we present, this method is used for registering an event handler that will be called when the text changes.

```
<%@ Page CodeFile="Ping.aspx.fs"
         Inherits="Demos.Ping" %>
<html>
<body>
  Search for: <fwc:TextBox runat="server" id="txtInput" />
  <div id="result">
    <fwc:Element runat="server" id="ctlOutput" />
  </div>
</body>
</html>
```

Figure 28. *Ping.aspx* file with the declarative markup for a sample page.

6.5.2 Building New Controls

We extend the ASP.NET programming model not only for writing pages, but also for writing new controls, so the users can define their set of controls, purely using our F# Web Toolkit and use them to build their web applications. Similarly to ASP.NET we allow building *user controls* and *custom controls*. The former is a control which is written in a similar style as a page, meaning that it consists of a file declaring markup (`Control.ascx`) and a file declaring the interaction logic (`Control.ascx.fs`). User controls are particularly useful when wrapping some complex functionality created using several elementary components into a block that can be used in multiple pages of the application. The second kind of control (custom controls) are elementary controls that are written as a single F# type. These controls can usually generate the HTML code programmatically, and handle updating the code on the client-side using the client-side DOM framework described in §6.2.3.

Since the components are written as mixed types, it is very easy to see what functionality of the control is available on the server-side and what functionality can be accessed on the client-side. The following example presents part of the interface of the `Calendar` control (used in the Lecture Organizer case study):

```
type Calendar =
  member SelectedDate       : NeutralM<DateTime> with get, set
  member HighlightDates     : ServerM<HighlightDatesDelegate>
  member ClientDateSelected : ClientM<ClientEvent>
```

As we can see, the `SelectedDate` member, which is used for setting or getting the selected date in the calendar can be used on both client-side and server-side, the `HighlightDates` member is an event that is invoked on the server-side when rendering a calendar, and the user can use it to return a list of highlighted days for the currently rendered month. Finally, the `ClientDateSelected` member is an event raised on the client-side when a date is selected and this is the most innovating part of our approach, because the event raised on the client-side can be handled by other client-side code without calling the server-side, but while preserving the same programming style.

Chapter 7
Key Language Features Used

In the last chapter we'd like to review all important aspects of the presented work and describe what language features were used to implement these aspects. The purpose of this chapter is to recapitulate novel concepts in our work as well as to summarize what has the programming language support to enable developing similar web development framework as is our F# Web Toolkit.

7.1 Heterogeneous Execution

Heterogeneous execution of homogeneous code is becoming more and more important as programming tools try to solve the language impedance mismatch and target more execution environments by a single language. Enabling smooth heterogeneous execution of the code written in a single programming language requires according to our experience with the F# Web Toolkit project following:

1) First requirement is to provide some way for accessing some form of AST of the code that should be executed in a non-standard way. This is in fact more a compiler feature enabled by the `ReflectedDefiniton` attribute and it doesn't require any explicit support in the programming language, though it can be nicely used in association with meta-programming support as in F#.
2) Supporting a scenario where the source code is compiled and distributed among several environments that execute it natively (e.g. part on the server using .NET and part on the client using the Silverlight platform) is slightly more difficult (and still unsolved in F#), because it may need to compile part of the source against different version of standard libraries and so the splitting would have to be done before the actual compilation.
3) Finally, it is essential to have a mechanism for verifying that the code executed in a non-standard way (by translating the AST to other language) satisfy all additional limitations of the target environment (e.g. uses only supported language constructs and accesses only types and functions supported by the target environment). How this mechanism could be implemented for F# is outlined in §7.5.1.

The need for heterogeneous execution can be demonstrated by a large number of examples including execution on the GPU and SQL mentioned in [10] and execution in web browser (either as a JavaScript or using a Silverlight platform) presented in our work. An additional example may be translating larger blocks of code to an SQL and "compiling" them to a stored procedure. In our syntax using monads it would look like following:

```
[<ReflectedDefinition>]
let countLines(db, country) =
  sql { let rows =
          { for c in db.Customers
            when c.Country = country
            -> c }
        return (count rows) }
```

We believe that a variant of the `ReflectedDefinition` attribute from F# would be relatively easy to implement in other languages as well and together with an alternative approach to annotating execution environment (mentioned in §5.1) could form an adequate support for the heterogeneous executions for example in languages like C#. A possible example of this approach is demonstrated in Figure 29. The other important part of support for heterogeneous execution, tracking of the modalities in a type system, requires however additional efforts.

```
[RunAt(Side.Client), ReflectedDefinition]
void PingServer()
{
  string str = this.txtInput.Text;
  string resp = this.ServerEcho(str)
  Window.Alert("Response: " + resp);
}
```

Figure 29. Possible use of the `ReflectedDefinition` attribute in C#.

7.2 Non-Standard Computations

In our project we use the F# monadic syntax to represent two kinds of non-standard computations. We use it to track the modality of the code in a type-system and for writing asynchronous computations on the client-side akin to the asynchronous workflows in F#.

The first use of monadic syntax actually uses only one aspect of monads, which is the ability to wrap a computation in a type specifying some additional properties of the computation and keep track of these additional properties in a type system. The use of monads only for this purpose may be problematic, because they raise some additional restrictions to the code that are not important for our code. These limitations arise for example when writing a sequence of applications where every application has a type `‘a -> M<’b>` and so it is impossible to use the natural dot-notation:

```
// Non-monadic code
let res = this.Foo().Bar().Result

// Monadic code
m { let! t1  = this.Foo()
    let! t2  = t1.Bar()
    let  res = t2.Result }
```

In a project focusing only on heterogeneous execution we would prefer using some weaker language construct for tracking the modality. One possible alternative may be Idioms presented in [37], however even Idioms are too strong for expressing just a modality of the code.

The second use of monadic syntax is rather standard use of monads for writing the code in a continuation-passing style, aside from the fact that we use it in the client-side code, however supporting this requires only a little effort in the F# to JavaScript translator and in fact it could be written without any explicit support, because the monadic syntax is stored in a desugared form in the F# quotations.

This programming style can be generally achieved only in a programming language that supports some form of monads. The use of asynchronous code isn't essential for implementing a project like this, but we believe that it is very important for giving better client-side programming model, since it partially allows dealing with the inversion of control in reactive applications. For example drag and drop operation can be nicely expressed using our `client_async` code (with some additional functions that are not currently present in our project):

```
let dragDrop(ctrl) =
  let rec dragging(pos) =
    client_async
     { match WaitForAnyEvent [ctrl.MouseUp; ctrl.MouseMove] with
       | MouseUp _    -> do! waiting()
       | MouseMove e -> do  ctrl.Location <- e.Postion - pos
                        do! dragging() }
  and waiting() =
    client_async
     { let! e = WaitForEvent(ctrl.MouseDown)
       do!  dragging(e.Position - ctrl.Location) }
  waiting()
```

7.3 Members as Symbols

Another interesting language feature that we use is the ability to work with a member (for example method) as a symbol. By symbol we mean some representation of the member that can be passed as an argument to a function and processed in other ways, but not necessarily executed. F# doesn't have any direct support for a notion of "symbol", but it can be suitably achieved by meta-programming, specifically by the ability to quote F# code. Let's look again at the example we presented earlier:

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

In our implementation, the Unicode quotes process the enclosed code as a quotation, but in an essence we just need to get a symbol representing the `set_ClientData` member.

The notion of "symbols" in a sense we use it here is common to some languages that employ dynamic typing[28], but as far as we are aware, the support of symbols with similar elegance as in these dynamic languages is not available in any strongly typed language.

[28] For example the Ruby language or the Smalltalk language

Conclusion and Related Work

7.4 Related Work

The two key areas that we focused on in this work are restricted development environment for writing the client-side code (including a limited choice of the programming languages and the lack of standard libraries) and the need to bridge the gap between client and server when writing a code of which the execution is distributed between these two environments including the difficulty of wrapping such code in reusable components. Finally, the last area that appeared as a relevant, although focusing it wasn't primary intention of this work, is finding a more adequate programming model for the development of web applications.

7.4.1 Rich Client-Side

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 approach to replace JavaScript completely was used in the Silverlight project [15], which tries to build a plug-in that provides a general purpose programming environment (based on the .NET Framework) for the client-side. The goal of the project is to make the plug-in compatible with most of the frequently used web browsers and platforms, which is essential for the usability of any project taking this approach. The project is relatively new, but once it becomes more broadly available it can be viewed as an attractive alternative to JavaScript as an execution environment. Nevertheless other web development problems focused in our work still remain, so it would be indeed interesting to adapt our work to use Silverlight as a client-side execution environment.

The extensions to the JavaScript 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 (including class-based OOP) or in a part of [26], which provides functions for writing functional reactive programs for the client-side. This way of extending client-side environment requires the least effort, but it doesn't fundamentally improve the developer experience. It also requires additional developer knowledge of the specific library. On the other side these extensions are extremely useful for the third possible solution – translating another language to JavaScript and we use part of the ASP.NET AJAX extension [7] for executing class-based OOP code generated from F# in JavaScript.

An alternative way of extending the client-side environment (purely in JavaScript) is to develop an embedded language (typically XML-based) and a processor of such language. This approach is used in commercially used frameworks such as ASP.NET AJAX [7] or Backbase [12]. In these frameworks, the XML-based language is used for declarative description of interactions between client-side components. Its advantages are that the XML declarations are easier to write and can be also verified for correctness to some point, but are limited in what interactions they allow users to define and also require knowledge of domain specific (XML based) language that is specific for every web framework.

The third approach is to implement a compiler to JavaScript. Such compiler can work with either a low-level language (like .NET IL code in [4]), or with a high-level language (for example Java in [5], C# in [6] or a Links language in [3]). Indeed, translating a low-level code (or implementing a back-end for existing compiler) looks very appealing, but the implementation can be very complex and in addition some higher level language features that could be translated directly to JavaScript (e.g. first-class functions and closures) may be already lost in the low-level code. The overall complexity of our implementation stack is very low in comparison to these approaches: our entire translator and library mappings consist of approximately 4,000 lines of F# code and only a handful of lines of bespoke JavaScript.

7.4.2 Client-Server Gap

The second problem of the web development 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 a more difficult problem and there are only a few projects that attempt to solve it to some extent (e.g. Volta [4] and Links [3]).

Solving the client-server gap and the language impedance mismatch 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. In the .NET environment, the possibilities for integration of the data access into a language 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]. In our work we integrate client-side and server-side code, though the calls between different execution environments are explicit, which we find rather useful, because we believe that the programmer should be aware of the presence of non-standard calls in his code. Conversely, in Links [3] the entire code is compiled to a continuation passing style code and calls between different environments are allowed implicitly. Another project that allows tight integration between execution environments is the HOP language [21].

Other related projects focus mostly only on the language impedance mismatch (e.g. [4, 5, 6]) and don't provide any direct integration between client-side and server-side code. The cross-environment calls are usually possible thanks to RPC or Web Services for invoking server-code from the client side. In Volta [4] the code is first written without explicit separation of environments (which also allows comfortable debugging support) and the explicit Web Service calls can be generated automatically.

7.4.3 Markup & Components

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 the authors of the Links project discuss several possible abstractions in [17].

7.4.4 Web Development Paradigms

The last aspect that deserves mentioning, though focusing on it wasn't primary goal of our work is overall the structuring of control-flow and data-flow in web applications. Our work follows the same path as most of the commercially used frameworks (e.g. [9, 29, 36]) and use the page-based model with event-based control flow, however there are some interesting alternatives.

Projects based on functional programming often use the continuation based model (for example in [31] or to some point in [3]) which allows dealing with the inversion of control [28] and makes the code more readable, though arguably some important aspects of the architecture are hidden when using this abstraction. Finally functional reactive programming gives a very promising approach and was successfully used in [26] for developing client-side part of the application.

7.4.5 Summary

The following table shows a summary of several possible approaches and the projects that followed them. Our project is displayed in the table under the name "F# Web Toolkit". In the table we compare programming languages that can be used for writing a code on the client and on the server, the runtime environment used on the client-side, the level of integration between client-side and server-side code and also the ability to use declarative markup to write the HTML code or even build an application using components.

Project	Client Language	Server Language	Client Runtime	Integrated Code	Declarative Markup§
MS AJAX [7]	JS‡	Any	JS	no	client/server
Script# [6]	C#	Any	JS	no	Server
GWT [5]	Java	Any	JS	no	None
Volta [4]	C#/VB/.NET†	.NET	JS	no	None
Silverlight [15]	.NET	Any	.NET	no	client/server
haXe [30]	haXe	haXe	JS	no	html/server
Links [3]	Links	Links	JS	yes	none£
F# WebToolkit	F#	F#, ASP.NET	JS	yes	Integrated

† Language support depends on the completeness of the decompiler from the IL code.
‡ JavaScript with several extensions that emulate .NET programming patterns.
§ In this column *client/server* means that some declarative code can be used on both sides, but the languages differ, *html* means that it is possible to use HTML language for writing the code, *none* means that whole page has to be constructed programmatically and *integrated* mans that the same style can be used on both sides.
£ Links makes this easier by supporting XML literals in the language directly.

7.5 Future Work

7.5.1 Improving Compile-Time Verification

In the section §6.3 we informally described a set of 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 the monadic typing. Even though monadic typing solves many of the correctness problems it can't be used everywhere and in some places we need an additional verification mechanism.

We believe that it would be feasible to implement such mechanism as an extensibility point in the F# language compiler, so developers of embedded languages in F# could implement custom rules that have to be checked in order to catch all possible errors during compile time.

In general we think that the compiler extensibility like this would be very helpful in any meta-programming scenario where the code is executed in heterogeneous environments and where additional *ad hoc* restrictions exist. It would be beneficial to use mechanism like this in most of the examples presented in [10], both translation of the F# code to SQL (when accessing database) and translation of F# language to GPU code working with matrices have some restrictions that are verified at runtime in current implementation.

7.5.2 Other Client-Side Runtime Environments

Other client-side environments could be used for two different purposes. On one hand it could be used to achieve better support for debugging of the client-side code (which is a difficult task in our current implementation). We could implement a mechanism similar to the one presented in [4] or [5] where during debugging, the client-side code is executed natively instead of being translated and executed in the browser and so the code can be debugged using any debugging tools for .NET/Java. A similar solution would be applicable to our project.

On the other hand we would like to investigate the possibility of targeting the Silverlight platform [15] as an alternative to the JavaScript for execution of the client-side code. Silverlight is interesting mainly because it may be available in most of the web browsers on most of the platforms in the near future (the implementation for the Mono platform [25] was also announced). It is easy to imagine that the support for this could be added to our toolkit as an additional modality with a new monadic type. Code written using this modality could be then executed natively on the client-side. Silverlight provides an environment rich enough to execute most of the F#/.NET code, though some of the functionality may not be available so it would be still very useful to have a mechanism for verifying that the client-side of the web application uses only functionality available in the Silverlight. This verification could be implemented as an complier extension using the mechanism proposed above.

7.5.3 Control and Data Flow in Web Applications

Our work so far was mostly focused on the integration of the client and the server-side of a single web application, so once this is satisfactorily solved, the next logical step of our work is to examine alternative and possibly better programming models for developing web applications as a whole.

The most appealing approach for future direction of the project seems to be the functional reactive programming, because the web applications can be indeed treated as a reactive system where many data structures (and not only those present on the client-side) are time-varying values (called *behaviors* in [33]) and the application is driven by events, which is the second common aspect of all functional reactive systems.

Also working with a slightly higher abstraction can eliminate some additional inconsistencies between the client and the server-side. For example the fact that server-side can't call client-side code is nicely eliminated in Link [3], which makes it possible to call a client-side code from a server-side thanks to the use of continuation-passing style. Though we may not use exactly this principle, it exhibits the overall goal to make the integration smooth.

7.5.4 Security and Data Validation

In §5.5 we discussed a security aspects of Ajax web applications in general as well as of our project. The problem that causes most of the security issues of Ajax applications is that the data from the client-side cannot be trusted in any way and so all verifications done on the client-side have to be performed explicitly again on the server-side. This is indeed caused by design of the web applications however we believe that it should be possible to provide a better abstraction for data that need to be verified and perform the re-verification on the server-side automatically when receiving a request from the client.

One possible solution for this would be to place the code performing the verification not in the client-side and server-side code, but as a property of the data type. The code performing the verification should be of course written as a neutral code which will allow the system to execute it from both client-side and server-side.

7.6 Conclusion

In this work we have shown how F# language 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 limitations of the client-side environment (when targeting JavaScript) including the lack of type-checked execution. We also gave an overview of all the relevant F# language features that we used in our work to make our solution easily reproducible with any other programming language that could support such extensions.

We use non-intrusive meta-programming in F# 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. 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#.

Finally, we combine two approaches for integrating client-side and server-side code (or code executing in multiple environments in general). We allow the usage of either F# attributes to annotate that a large block of code (module or a class) has specific modality or the usage of monadic modalities to separate members intended to run in specific environments. Thanks to the typing properties of monads the calls between different environments are made explicit, which prevents the users from writing an incorrect code and also gives a clue where the application performs an inefficient call between environments. In addition we use subtyping between monadic types to enable implicit calls between blocks of code with different, but compatible modalities.

From a web development perspective, we allow the client-side code to be written in an asynchronous way similarly to the continuation-passing style presented in [17], but we make this explicit using F# monadic syntax. This enables us to use continuation based model only when it is appropriate without losing the possibility to write code in a standard way when needed. We also demonstrate a way to build composable components based on those used in ASP.NET that allow wrapping of both server-side and client-side functionality. These components expose a clear interface where a modality of the exposed member is tracked by a type system that makes it extremely easy to understand the exposed functionality. Finally, thanks to our homogeneous programming style it is possible to express both server-side and client-side interaction between components in a uniform way.

The presented aspects of our work have 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 the F# Web Toolkit becomes the world's biggest web-development platform or not.

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 Lightweight Language Extensions. 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# Web Toolkit 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. In *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] The Mono Project. 2007. See http://www.mono-project.com

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

[27] David Tarditi, Sidd Puri, Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006

[28] Christian Queinnec. Inverting back the inversion of control or, Continuations versus page-centric programming. Sigplan Not., 2003

[29] Ruby on Rails. 2007. See http://www.rubyonrails.org

[30] Nicolas Cannasse. haXe: A Web-oriented Programming Language. OSCON 2006. See: http://ncannasse.free.fr/files/haxe_oscon2006.pdf

[31] C. Queinnec. Continuations to program web servers. ICFP, 2000.

[32] Philipp Haller, Martin Odersky. Event-based Programming without Inversion of Control. In *Proceedings of JMLC*, LNCS, pages 4-22. Springer 2006.

[33] Gregory Cooper, Shriram Krishnamurthi. FrTime: Functional Reactive Programming in PLT Scheme. Tech. Rep. CS-03-20, Brown University

[34] Martín Abadi, Luca Cardelli. A theory of objects (book). Springer-Verlag 1996. ISBN 0-387-94775-2

[35] Philip Wadler. Monads for Functional Programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming* in *Proceedings of the Båstad Spring School*, May 1995, Springer Verlag Lecture Notes in Computer Science 925.

[36] Sun Microsystems Inc. Java Server Pages Technology, 2007. See: http://java.sun.com/products/jsp/

[37] Conor McBride. Ross Paterson. FUNCTIONAL PEARL: Idioms: applicative programming with effects. In *Journal of Functional Programming*. Cambridge University Press, 2007.

Appendix
Contents of the Attached CD

The attached CD contains the following folders:

- **source** – This directory contains the source code of the F# Web Toolkit project. The subdirectory `WebToolkit.Core` contains implementation of the core components of the project including the F# to JavaScript translator and the components for client-server integration. The subdirectory `WebToolkit.Controls` contains basic set of controls, including those used in the presented case-studies.
- **fsharp** – This directory contains the installation of the F# compiler in a version that is compatible with the attached source code.
- **demos** – This directory contains the three case-studies presented in this work (WSH Scripting, Symbolic Manipulations and Lecture Organizer) as well as one additional case-study (Ajax Dictionary) and also a database with data used by the Dictionary and Lecture Organizer samples.
- **documents** – Contains the text of this thesis in a PDF format.