Ajax-style Client/Server Programming with F#

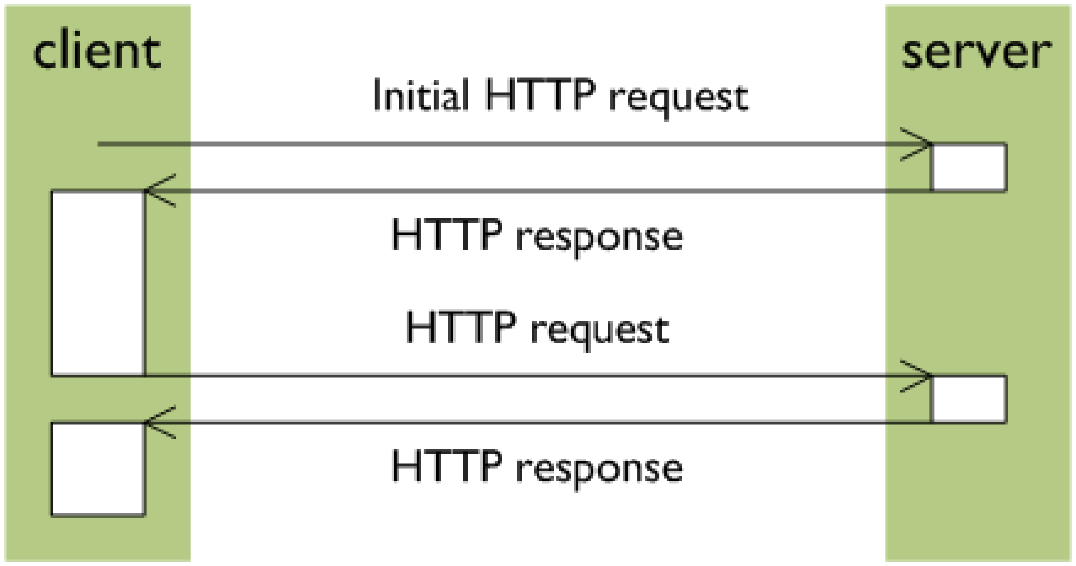
Tomáš Petříček & Don Syme

Why are Web Applications Hard?

- Limited client-side environment
  - Only JavaScript (no client-side installation allowed!!!)
- Discontinuity between client and server side
  - Two parts of the same application!
- Components in web frameworks are only server-side
  - No standard way for expressing client-side interactions

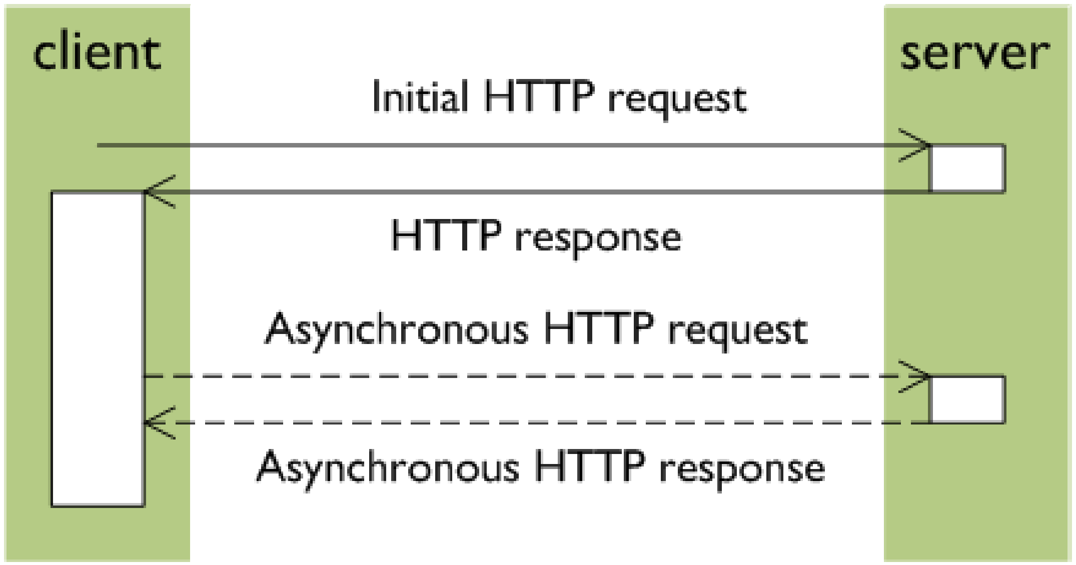
Ajax-style Web Control Flow

- HTTP protocol is based on request-response principle



Ajax-style Web Control Flow

- HTTP protocol is based on request-response principle



- Ajax-style applications update the page dynamically using asynchronous requests

What’s going on?

- Program is “authored” on the server-side
  - But runs primarily on the client-side
  - No state on the server (except during callbacks)

- Program is “authored” in server-side language (e.g. Visual Basic or PHP)
  - But the main point of this program is to generate a client-side program in JS
  - Executing some (most) code as JavaScript is unavoidable

- Summary
  1. Client asks server for page
  2. Server gives back JavaScript
  3. Client runs JavaScript
  4. Client calls back server asynchronously
  5. Server talks to database during callbacks
  6. Server responds with updated DOM/JavaScript for client

Project Aims

1) Single programming language - F#
2) Type-checked client & server code
3) Modality-checked client & server code
4) Be Realistic
   - Integrate with ASP.NET (or Apache …)
   - No client-side installation required
   - Look forward (SilverLight, …)

Related Work

- **Volta** (Erik Meijer et. al., Microsoft)
  - Client side in .NET languages (by translating IL to JavaScript)
- **Links Language** (Philip Wadler et. al., University of Edinburgh)
  - Single language for all three tires (client/server/database)
- **Google Web Toolkit**
  - Client side in Java (by translating Java to JavaScript)
- **Silverlight** and **Flash**
  - Richer client-side execution frameworks
- **Flapjax** (Brown University PLT)
  - Functional Reactive Programming on the client side

Translating F# to JavaScript

- Translator needs to solve the following problems:
  1) Translating the F# language
  2) Accessing native JS functionality
     - Manipulating with DOM, calling native JS functions
  3) Using standard .NET/F# types
     - Working with standard data types, collections, etc…

SAMPLE: Symbolic manipulation (I.)

Not-so-simple F# application running in web browser

SAMPLE: Pattern matching in F# and JS

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

SAMPLE: Pattern matching in F# and JS

```
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 varfunc(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(), [eval(a), eval(b)]);
          var ea = t.Get(0);
          var eb = t.Get(1);
          return ea + eb
        } else {
          return Lib.Utils.FailWith("unknown op.");
        }
      }
    }
  })
  return eval(nd);
}
```

Translating F# to JavaScript

- Problem #2 – Accessing native JS functionality

```
let Test() =
    Window.Alert("Hello world!")
    ...
```

  - Using mock types and mapping attributes:

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

Translating F# to JavaScript

- Problem #3 – Using standard .NET/F# types

```
let Test() =
  let s = new StringBuilder()
  ignore(s.Append("Hello world!"))
s.ToString()
```

  - Re-implementing standard libraries in translatable F#:

```
[<ClientSide; ExternalType(type System.Text.StringBuilder)>]
type StringBuilder() =
  let strings = [| |]
  member this.Append(s:string) =
    ArrayJS.Add(this.strings, s)
    this
  member this.ToString() =
    ArrayJS.Join(this.strings, "")
```

Integrating client and server-side code

- Using monads for representing modality of the code
  - Very appealing typing properties – checks modality of the calls
  - Gives us control over the execution

```
//: unit -> ClientMonad<string>
member x.GetMessage() =
 client { -> "Hello world!" }

//: unit -> ClientMonad<unit>
member x.ShowAlert() =
 client { let! s = x.GetMessage()
          do   Window.Alert(s) }

//: unit -> ServerMonad<unit>
member x.ServerFoo() =
 server { ... }
```

Calling server-side from client-side

- Using ‘serverExecute’ function and ‘async’ client code
  - **serverExecute:** ServerM<T> -> ClientAsyncM<T>
  - ‘async’ code translated to continuation-passing style

```
member x.Ping(s) =
 server
  { -> String.rev s }

member x.ButtonClick() =
 client
  { do! x.ShowLoading()
    do! asyncExecute
         (client_async
           { let  arg = x.txt.Text
             let! res = serverExecute(x.Ping(arg))
             do!  x.label.set_Text(res)
             do!  x.HideLoading() }) }
```

Asynchronous client-side code

- Very useful for expressing some common patterns
  - For example polling the server for updates

```
member this.RefreshImage(last) =
  client_async
    { do! Async.SleepAsync(300)
      if (last <> this.txt.Text) then
        let! url = serverExecute
                    (this.GenerateImage(this.txt.Text))
        do!  this.imgPlot.set_ImageUrl(url)
      do! this.RefreshImage(this.txt.Text) }
```

Asynchronous client-side code

- Very useful for expressing some common patterns
  - For example polling the server for updates

- What is ‘client_async’ monad?
  - Implementation of continutation monad in (translatable) F#
  - It is possible to write other ‘async’ primitives

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

SAMPLE: Symbolic manipulation (II.)

JavaScript ‘async’ monad & calls to the server

Composable components for ASP.NET

- Calling a page event handler on the server-side:
  - We usually need to manipulate with more components
  - Some components may require executing server-side code
  - Component needs to update internal state and GUI

```
// Client invokes server-side code
member x.buttonClicked() =
 client
  { do! serverExecute(x.updatePage()) }

// Server-side code updates components
member x.updatePage() =
 server
  { let data = Database.LoadData()
    do! this.list.set_Data(data)
    do! this.calendar.set_Highlighted(hl) }
```

Composable components for ASP.NET

- State handling is property of the ‘server’ monad
  - Collects a list of client-side operations to execute

```
type Repeater =
  // Server-side code to update GUI
  member x.set_Data(data) =
   server
    { let data = data |> Seq.to_list
      do! <@! §x.set_ClientData(§data) !@> }

  // Executed on the client-side to view data
  member x.set_ClientData(data) =
   client
    { let html = (* generate html *)
      do  this.InnerHtml <- html }
```

  - ServerM<‘a> = ‘a * ServerUpdate list

SAMPLE: Lecture organizer

Using composable ASP.NET components

Summary

1) Single programming language - F#
   - Thanks to non-intrusive meta-programming capabilities

2) Integrated server & client-side code
   - Thanks to typing properties of monads

3) Client-side state updates in the server code
   - Thanks to F# monads & symbolic meta-programming

Thank you!

Questions, discussion, etc…

For more information:

- Web site with samples & more information:
  http://tomasp.net/fswebtools

- Shared source project at CodePlex:
  http://www.codeplex.com/fswebtools