In the Age of Web:
Typed Functional-First Programming Revisited

Tomas Petricek
University of Cambridge, UK
tomas@tomasp.net

Don Syme
Microsoft Research Cambridge, UK
don.syme@microsoft.com

Most programming languages were designed before the age of web. This matters because the web changes many assumptions that typed functional language designers take for granted. For example, programs do not run in a closed word, but must instead interact with (changing and likely unreliable) services and data sources, communication is often asynchronous or event-driven, and programs need to interoperate with untyped environments.

In this paper, we present how F# language and libraries face the challenges posed by the web. Technically, this comprises using *type providers* for integration with external information sources and for integration with untyped programming environments, using *lightweight meta-programming* for targetting JavaScript and *computation expressions* for writing asynchronous code.

In this inquiry, the holistic perspective is more important than each of the features in isolation. We use a practical case study as a starting point and look how F# language and libraries approach the challenges posed by the web. The specific lessons learned are perhaps less interesting than our attempt to uncover hidden assumptions that no longer hold in the age of web.

1 Introduction

Among the ML family of languages, F# often takes a pragmatic approach and emphasizes ease of use and the ability to integrate with its execution environments[1] over other aspects of language design. If you use the F# langage as ML, you get most of the good well-known properties of ML[2]. However, F# leaves enough *holes* that let you use it *not* as ML. This is the space that we explore in this paper.

This additional flexibility makes it possible to use F# in ways that break the common assumptions that are often taken for granted in languages such as ML and Haskell[3]. The focus on the web directs our inquiry and provides an angle for reconsidering such assumptions.

Perhaps the most remarkable assumption is the idea that programs fundamentally operate in a closed world. Although we have learned how to perform FFI and I/O [6, 14], those are treated as dealing with the "dirty real world". For a practical solution, we argue that we need to go much further, but deeper integration with (untyped) JavaScript libraries and (evolving) services inevitably breaks strict type safety requirements.

We do not claim that the F# approach is the only possible. Rather, this paper should be seen as a programming language experiment [11] or an empirical observation of the approach used by the F# community. We aim to provide an intriguing exploration of hidden assumptions and present what can be achieved using a combination of F# features. We do so by starting with a simple, yet real-world problem and then exploring a solution. More specifically, the contributions of this position paper are:

[1] Historically, this applied to the .NET runtime, but the same applies to integrating with JavaScript in the web context.
[2] For example, F# does not require type annotations when used as ML, but requires them when used with .NET objects.
[3] In a way, we are trying to uncover the hidden assumptions of the functional programming *research programme* that are normally "*rendered unfalsifiable by the methodological decisions of its protagonists*." (Lakatos [9] as quoted by Chalmers [4])

Submitted to:
ML 2014

- We present a case study (Section 2) showing how a combination of numerous F# language features can be used for the development of modern web applications. This is not a toy demonstration, but an example of how F# is used in the industry.
- We discuss how type providers make it possible to access external information sources in web applications (Section 3.1) and integration with (untyped) programming environments such as the JavaScript ecosystem (Section 3.2).
- We show how F# approaches the problem of compilation to JavaScript using a library called FunScript (Section 4.1), outlining important practical concerns such as interoperability (Section 4.2) and asynchronous execution (Section 4.3).
- Throughout the paper, we discuss how the age of the web breaks the assumptions commonly taken as granted in typed functional programming. We revisit the notion of type safety in the context of web (Section 3.3) and the notion of fixed language semantics (Section 4.4).

In the first part of the paper (Section 2), we present a case study of using F# for web development. The rest of the paper (Section 3 and 4) discusses the arising issues in more depth. The source code and running demo for the case study is available at: `http://funscript.info/samples/worldbank`

2 Case Study: Web-based data analytics

In this case study, we develop a web application shown in Figure 1, which lets the user compare university enrolment in a number of selected countries and regions around the world. The resulting application runs on the client-side (as JavaScript) and fetches data dynamically from the World Bank [1].

The application is an example of a web page that could be built in the context of data journalism [7]. As such, it is relatively simple, works with just a single data source and uses a concrete indicator and a hard-coded list of countries *i.e.* to illustrate a point made in an accompanying article.

2.1 Accessing World Bank data with type providers

To access the university enrollment information, we first obtain a list of countries using the World Bank type provider from the F# Data library [12]. The type provider exposes the individual countries as members of an object (the notation ``Country Name`` is used for identifiers with spaces):

```
type WorldBank = WorldBankData⟨Asynchronous = true⟩

let data = WorldBank.GetDataContext()
let countries =
  [ data.Countries.``European Union``
    data.Countries.``Czech Republic``
    data.Countries.``United Kingdom``
    data.Countries.``United States`` ]
```

The type provider connects to the World Bank and obtains a lisf of countries at *compile-time* and at *edit-time* (when using auto-completion in an editor). This means that the list is always up-to-date and we get a compile time error when accessing a country that no longer exists (a property discussed in Section 3.1).

On the first line, we provide a static parameter Asynchronous. Static parameters are resolved at compile-time (or edit-time). Here, we specify that the exposed types for accessing information should support only non-blocking functions. This is necessary for a web-based application, because JavaScript only supports non-blocking calls (using callbacks) to fetch the data.

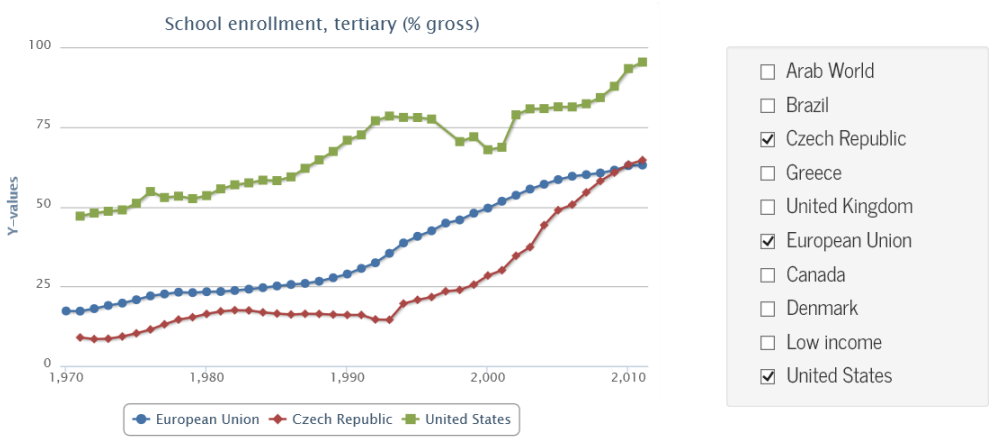

Figure 1: Case study – web application for comparing university enrollment in the world

2.2 Interoperating with JavaScript libraries

To run the sample application on the client-side we use FunScript [3], which is a library that translates F# code to JavaScript (Section 4). Aside from running as JavaScript, we also want to use standard JavaScript libraries, including jQuery for DOM manipulation and Highcharts for charting. FunScript comes with a type provider that imports TypeScript [10] definitions for JavaScript libraries:

```
type j = TypeScript⟨"jquery.d.ts"⟩
type h = TypeScript⟨"highcharts.d.ts"⟩
```

The `d.ts` files are type annotations created for the TypeScript language. Here, the type provider mechanism lets us leverage an existing effort for annotating common JavaScript libraries. The type provider analyses those definitions and maps them into F# types named j and h that contain statically typed functions for calling the JavaScript libraries (we will use them shortly). The file names are static parameters (same as Asynchronous earlier) and are statically resolved and accessed at compile-time or edit-time.

Importing types for JavaScript libraries into the F# type system has interesting implications, because the TypeScript language does not have the traditional type safety property [15]. We return to this topic in Section 3.2. Next, we generate checkboxes that appear on the right in Figure 1:

```
let jQuery command = j.jQuery.Invoke(command)

let infos = countries |> List.map (fun country →
  let inp = jQuery("<input>").attr("type","checkbox")
  jQuery("#panel").append(inp).append(country.Name)
  country.Name, country.Indicators, el)
```

To manipulate the DOM (Document Object Model), we are using the jQuery library in a way that is very similar to code that one would write in JavaScript. We define a helper function jQuery (hiding some of the complexities of the mapping) and use it to create the `"<input>"` element and specify its attributes. Note that members like append and attr are standard jQuery patterns. The compiler sees them as ordinary object members. When writing code using F# editors based on the F# Compiler Serivce [8], they also appear in the auto-complete list.

Although the jQuery library is not perfect, it is a de facto standard in web development. The FunScript type provider makes it possible to integrate with it painlessly without explicitly specifying any FFI interface and without manual wrapping (see also Section 4.2).

Note that we use a standard F# function List.map to iterate over the countries. This has a side-effect of creating the HTML elements, but it also returns a new list. The result is a list of string $*$ Indicators $*$ jQuery values representing the country name, its indicators (for accessing the World Bank data) and the created DOM object representing the checkbox.

2.3 Loading data and updating the user interface

The main part of the sample program is a function render that asynchronously fetches data for selected countries and generates a chart. To keep the code simple, we iterate over the infos list from the previous section and load data for countries one by one:

```
let render () = async {
  let head = "School enrollment, tertiary (% gross)"
  let o = h.HighchartsOptions()
  o.chart ← h.HighchartsChartOptions(renderTo = "plc")
  o.title ← h.HighchartsTitleOptions(text = head)
  o.series ← [| |]

  for name, ind, check in infos do
    if unbox⟨bool⟩ (check.is(":checked")) then
      let! v = ind.``School enrollment, tertiary (% gross)``
      let data = vals |> Seq.map (fun (k,v) → [| number k; number v |]) |> Array.ofSeq
      opts.series.push(h.HighchartsSeriesOptions(data,name)) }
```

Although the function looks like ordinary code, it is wrapped in the async {...} block, which is an F# computation expression [13]. The F# compiler performs de-sugaring similar to the CPS transformation and interprets keywods such as let! and for using special operations (monadic bind and other). The async identifier determines that we are writing asynchronous workflow [16] that makes it possible to include non-blocking calls in the block.

Here, the non-blocking call is done when accessing the ``School enrollment, tertiary (% gross)`` indicator using the let! keyword. The indicator is a member (with a name wrapped in back-ticks to allow spaces) exposed as an asynchronous computation by the World Bank type provider. The rest of the code is mostly dealing with the DOM and the Highcharts library using the API imported by FunScript – we iterate over all checkboxes and generate a new chart series for each checked country.

Two notable points here are that async translated to JavaScript is restricted to a single thread, which is not the case for ordinary F# code (Section 4.3) and that the HighchartOptions object preserves some of the underlying JavaScript semantics (Section 4.2). Finally, the last part of the example code registers event handlers that redraw the chart when checkbox is clicked:

```
for _,_, check in infos do
  check.click(fun _ → Async.StartImmediate(render()))
```

The click operation (exposed by jQuery) takes a function that should be called when the event occurs. Calling it is a side-effectful operation that registers the handler. As render is an asynchronous operation, we invoke it using the StartImmediate primitive from the F# library, which starts the computation without waiting for the result (the only way to start a non-blocking operation in JavaScript).

2.4 Larning from the case study

The case study shows that we can develop a simple interactive data visualization (that could be built, for example, by data journalists) in less than 30 lines of F# code. The code uses many typical functional patterns (lists, first-class functions, data types), but also uses features that are more specific to F# (type providers, objects, computation expressions).

Before analysing the interesting aspects of this case study, we briefly review the points that we find appealing and points that many would find unappealing or, at least, peculiar. First, the appealing points:

- The ML approach to types and type inference can be extended from (closed-world) data types to (open-world) types for rich information sources such as World Bank. The sample code is fully statically typed without explicit type annotations. Critically, types are also used for exploratory programming when finding indicators using auto-complete in an editor.
- The case study demonstrates that core ML programming style can be used in the context of client-side (JavaScript) web development. We used functional lists, standard higher-order functions such as `List.map` in much the same way as when writing ordinary F#.
- In addition to standard functional constructs, we were also able to reuse F# asynchronous workflows to write non-blocking code that requests data from a web service (World Bank), rather than using error-prone explicit callbacks that are common in JavaScript.
- Finally, we were able to painlessly call Highcharts and jQuery. No explicit wrapping or importing of individual functions and types was necessary. Moreover, despite the differences between the F# and JavaScript object model, the code is close to idiomatic F#.

Now, the following list looks at the aspects that appear unappealing or peculiar, especially when coming from the traditional functional programming background:

- The World Bank type provider lifts information about countries to the type level. As a result, we can easily write `data.Countries.``Czech Republic``, but if Czech Republic is removed from the World Bank (and becomes Czechoslovakia again), the code will no longer compile (Section 3.1).
- The TypeScript language is unsound due to covariant generics [15]. Thus importing types from TypeScript definitions introduces a potential unsoundness into the F# code (Section 3.2).
- When compiling F# to JavaScript, the FunScript library does not fully preserve the semantics of F#. For example, numerical types behave as in JavaScript (Section 4.1) and asynchronous workflows run on a single thread (Section 4.3).

The most notable observation about the above points is that there is often both a positive and a negative side: we can nicely access World Bank data, but it affects soundness properties; we can interoperate with JavaScript libraries, but we can not fully hide undesirable JavaScript behaviours.

The aim of this paper is not to make value judgements and argue what is better. Using the case study as a basis, we claim that the outlined approach is *one possible* and that it *works in practice*. In the rest of the paper, we give more details about the most important aspects of the approach and discuss alternatives‘’.

3 Type providers

a
b

c

3.1 Integrating data

3.2 Integrating languages

3.3 Relativized type safety

4 Compiling to JavaScript

4.1 Quotations

We have been doing this for long. You could use compiler backend or meta-programming. For JS, backend is perhaps better. For SQL, Freebase, CUDA, ... meta-programming is good.

This is never going to work fully – there are different exception mechanisms, different numbers, different casts (for example, unbox)

4.2 Libraries

We could ignore all JS libraries and reimplement everything (or run every Hackage library). This works for some scenarios, but not for the case study covered here. You need to have access to standard JS libraries.

This brings challenges though – because the libraries are not F# libraries. (for example, we have to initialize the array of series)

4.3 Async

JavaScript async is single-threaded. Should we have "js-async"? No, that would be mess, accept that the semantics is relativized.

4.4 Relativized semantics

5 Conclusions and Future Work

[5] [2]

5.1 Big Picture

Is there a grand theory? No!

Perhaps we could claim one, but it isnt there [Feyerabend] – and that is the point (leave holes).

5.2 Conclusions

References

[1] The World Bank (2015): *School enrollment, tertiary (% gross)*. Available at `http://data.worldbank.org/indicator/SE.TER.ENRR`.

[2] Joel Bjornson, Anton Tayanovskyy & Adam Granicz (2011): *Composing Reactive GUIs in F# Using WebSharper*. In: *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages*, IFL'10, Springer-Verlag, Berlin, Heidelberg, pp. 203–216. Available at `http://dl.acm.org/citation.cfm?id=2050135.2050148`.

[3] Zach Bray & Contributors (2015): *FunScript: F# to JavaScript with type providers*. Available at `http://funscript.info`.

[4] Alan F Chalmers (2013): *What is this thing called science?* Open University Press.

[5] Loïc Denuzière, Ernesto Rodriguez & Adam Granicz (2014): *Piglets to the Rescue: Declarative User Interface Specification with Pluggable View Models*. In: *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*, IFL '13, ACM, New York, NY, USA, pp. 105:105–105:115, doi:10.1145/2620678.2620689. Available at `http://doi.acm.org/10.1145/2620678.2620689`.

[6] Sigbjorn Finne, Daan Leijen, Erik Meijer & Simon Peyton Jones (1999): *Calling Hell from Heaven and Heaven from Hell*. In: *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, ACM, New York, NY, USA, pp. 114–125, doi:10.1145/317636.317790. Available at `http://doi.acm.org/10.1145/317636.317790`.

[7] Jonathan Gray, Lucy Chambers & Liliana Bounegru (2012): *The data journalism handbook*. O'Reilly Media, Inc.

[8] The F# Core Engineering Group (2015): *F# Compiler Services: Editor services*. Available at `http://fsharp.github.io/FSharp.Compiler.Service/editor.html`.

[9] Imre Lakatos (1970): *Falsification and the Methodology of Scientific Research Programmes*. In Imre Lakatos & Alan Musgrave, editors: *Criticism and the Growth of Knowledge*, Cambridge University Press, pp. 91–196.

[10] Microsoft & Contributors (2015): *TypeScript*. Available at `http://typescriptlang.org`.

[11] Tomas Petricek (2014): *What can Programming Language Research Learn from the Philosophy of Science?* In: *Proceedings of the 50th Anniversary Convention of the AISB*.

[12] Tomas Petricek, Gustavo Guerra & Contributors (2015): *F# Data: Library for Data Access*. Available at `http://fsharp.github.io/FSharp.Data/`.

[13] Tomas Petricek & Don Syme (2014): *The F# Computation Expression Zoo*. In: *Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages - Volume 8324*, PADL 2014, Springer-Verlag New York, Inc., New York, NY, USA, pp. 33–48, doi:10.1007/978-3-319-04132-2_3. Available at `http://dx.doi.org/10.1007/978-3-319-04132-2_3`.

[14] Simon L. Peyton Jones & Philip Wadler (1993): *Imperative Functional Programming*. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, ACM, New York, NY, USA, pp. 71–84, doi:10.1145/158511.158524. Available at `http://doi.acm.org/10.1145/158511.158524`.

[15] Aseem Rastogi, Nikhil Swamy, Cedric Fournet, Gavin Bierman & Panagiotis Vekris (2014): *Safe & Efficient Gradual Typing for TypeScript*. Technical Report MSR-TR-2014-99, Microsoft Research. Available at `http://research.microsoft.com/apps/pubs/?id=224900`.

[16] Don Syme, Tomas Petricek & Dmitry Lomov (2011): *The F# Asynchronous Programming Model*. In: *Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages*, PADL'11, Springer-Verlag, Berlin, Heidelberg, pp. 175–189. Available at `http://dl.acm.org/citation.cfm?id=1946313.1946334`.