Doing web-based data analytics with F#
(Case study)

Tomas Petricek
University of Cambridge
tomas.petricek@cl.cam.ac.uk

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

Synopsis

With type providers that integrate external data directly into the static type system, F# has become a fantastic language for doing data analysis. Rather than looking at F# features in isolation, this paper takes a holistic view and presents the F# approach through a case study of a simple web-based data analytics platform.

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 environment(s) over other aspects of language design. It has a number of interesting features that follow this design principle:

- ***Type providers*** [1] are compiler and editor extensions that integrate external data sources into the language by lazily providing types for accessing specific data sources.
- ***Asynchronous workflows*** [2] is a library for writing asynchronous (non-blocking) code similar to Lwt [3] based on the computation expression syntax [4].
- ***Quotations*** [5] provide a modest form of meta-programming designed to simplify interoperability with other execution environments such as SQL, CUDA or JavaScript.

We present a case study that demonstrates how these features come together when building a web-based data visualization.

This contribution should be seen as a "programming language experiment" [6]. We hope to provide an intriguing exploration of what can be achieved by the combination of F# features when solving a simple, yet real-world problem. Full source code for is also available at: http://funscript.info/samples/worldbank

Case study

As an example, we aim to develop a simple modern web application shown in Figure 1 for comparing university enrolment in a number of selected countries and regions around the world. The resulting application should run on the client-side (as JavaScript) and should fetch data dynamically from the WorldBank.

We use FunScript [8] which is an F# library that takes a quotation of a program and translates it to JavaScript. To manipulate DOM and charts, we use jQuery and Highcharts (standard JavaScript libraries). To access those in a type-safe way, FunScript has a type provider that imports TypeScript [9] definitions:

```
type j = TypeScript<"jquery.d.ts">
type h = TypeScript<"highcharts.d.ts">
```

The `d.ts` files are type annotations created for the TypeScript language. The type provider analyses those and maps them into F# types named j and h that contain statically typed functions for calling the JavaScript libraries (we'll use them shortly).

The file names are specified in angle brackets (akin to generic type parameters), because they are statically resolved. The type provider generates the types at compile-time.

The next step is to obtain a list of countries. This is done using the F# Data type provider for WorldBank [7]:

```
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`` ]
```

When loaded in the compiler, the type provider connects to the WorldBank via a REST API and obtains a list of countries. 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.

The static parameter `Asynchronous` specifies that the exposed types for accessing country information should be only non-blocking. This is necessary for web-based application, because JavaScript only supports non-blocking calls to fetch the data.

Now we generate checkboxes (on the left in Figure 1) by creating a new `<input>` element and adding it to the DOM:

```
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)
```

We are using the standard jQuery library to manipulate DOM. Although this library is not perfect, it is de-facto standard in web development and the FunScript type provider makes it possible to integrate with it painlessly. We define a helper `jQuery` and use it to create the elements. Note that members like `append` and `attr` are standard jQuery patterns and are included in the auto-complete list when writing F# code using editors such as Emacs, MonoDevelop and Visual Studio.

The result is a list of `string * Indicators * jQuery` values representing country name, its indicators and DOM object representing the check-box. The main part of our program is a `render` function that asynchronously fetches data for checked countries and generates a chart:

```
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, tert. (% gr.)``
      let data = convertValues(v)
      h.HighchartsSeriesOptions(data, name)
      |> opts.series.push }
```

Although the function looks like ordinary F#, it is wrapped in the `async { .. }` block, which denotes that it is non-blocking. The

F# compiler performs de-sugaring similar to CPS transformation, which makes it possible to include non-blocking calls in the code. Here, the non-blocking call is done when accessing the ``School enrollment, tert. (% gr.)`` 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 WorldBank 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 series for each checked country. Finally, the last part of the code registers event handlers that re-draw the chart when checkbox is clicked:

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

The `click` operation (exposed by jQuery) takes a function that should be called when the event occurs. Because `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.

Observations

In the limited space, we have not explained every single detail of the sample program. Here, we look at some of the interesting aspects of the development.

Type providers for data access. The sample uses a type provider that provides types specifically for the WorldBank and so we can access countries and indicators as members. The F# Data [7] library also includes type providers that infer the types from a sample JSON and XML documents and can be used to call arbitrary REST-based web services.

Type providers for integration. Type providers are not limited to data access. Here, we used the `TypeScript` provider that imports type definitions for JavaScript libraries and makes it possible to call them easily (via meta-programming). In other contexts, type providers have been used to provide access to libraries of the statistical language R and Matlab.

Asynchronous workflows. In web browser, calls to retrieve data from services such as WorldBank have to be done via a callback (to avoid blocking the browser). This *inversion of control* makes it difficult to express standard control-flow structures such as `for` loops. Asynchronous workflows are built on top of F# computation expressions which provide a familiar syntax for non-standard (e.g. asynchronous) computations.

Meta-programming. The program implemented in the previous section is not executed as-is. Instead, the FunScript library takes the code as an F# quotation and translates it to JavaScript. The translation is done after the compiler de-sugars the `async { … }` block and the types generated by type providers. Thus, the translation only needs to handle primitive library constructs (such as `Bind` and `Return` operations of asynchronous workflows and the primitives for working with JSON used by `WorldBankData`).

Conclusions

This paper presented a number of features available in F# in the context of the development of a web-based interactive data analysis tool. Rather than focusing on technical details of individual language features (and comparing them with other languages), we used a more holistic view.

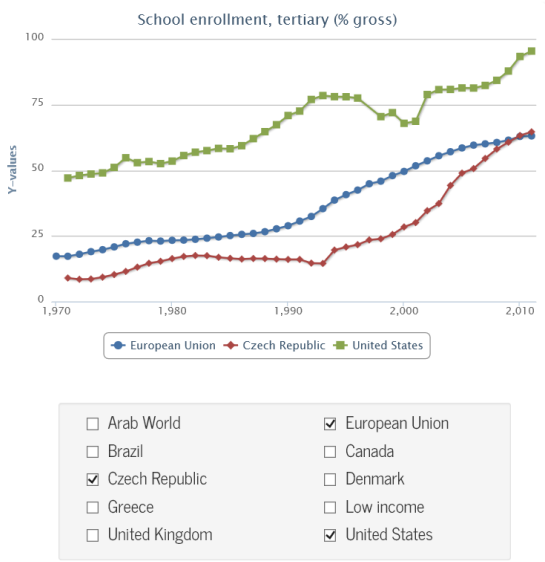

Figure 1. Comparing university enrollment in EU, US and CZ.

The presented case study should be seen as an experiment demonstrating what is enabled by the combination of type providers, asynchronous workflows and light-weight support for meta-programming. Our experience with the development of the presented case study suggests that:

- Integration with external data sources (e.g. WorldBank) and external execution environments (such as jQuery and JavaScript in general) cannot be overstated.
- Light-weight and non-intrusive syntactic extensions (e.g. asynchronous workflows) and meta-programming capabilities that integrate well with modern editors (e.g. autocomplete) contribute to the ease of development.

Although a case study such as this one focus on subjective observations rather than "hard" scientific facts, it is our hope that the presented example demonstrates the power of the ML family of languages from a novel, and perhaps a slightly different, angle.

References

[1] Syme, Don, et al. "Strongly-typed language support for internet-scale information sources." Technical Report MSR-TR-2012-101, Microsoft Research, 2012.

[2] Syme, Don, Tomas Petricek, and Dmitry Lomov. "The F# asynchronous programming model." In Proceedings of PADL, 2011. 175-189.

[3] Vouillon, Jérôme. "Lwt: a cooperative thread library." Proceedings of ML Workshop, 2008.

[4] Petricek, Tomas, and Don Syme. "The F# Computation Expression Zoo." In Proceedings of PADL, 2014.

[5] Syme, Don. "Leveraging .NET Meta-programming Components from F#." Proceedings of ML Workshop, 2006.

[6] Petricek, Tomas. "What can Programming Language Research Learn from the Philosophy of Science?" Proceedings of AISB 2014

[7] "F# Data: Library for Data Access". Available at: http://fsharp.github.io/FSharp.Data

[8] "FunScript – F# to JavaScript with type providers". Available at: http://funscript.info

[9] "TypeScript." Available at: http://www.typescriptlang.org