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 world, 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 the 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 targeting 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 at 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# language 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 [11, 23], 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 one. Rather, this paper should be seen as a programming language experiment [19] 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 [15], quoted by Chalmers [5])

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 industry.
- We discuss how type providers make it possible to access external information sources in web applications (Section 3.2) and integration with (untyped) programming environments such as the JavaScript ecosystem (Section 3.3).
- 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 for granted in typed functional programming. We revisit the notion of type safety in the context of the web (Section 3.4) 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 enrollment 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 [13]. 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 [20]. 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 list 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 3.2).

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.

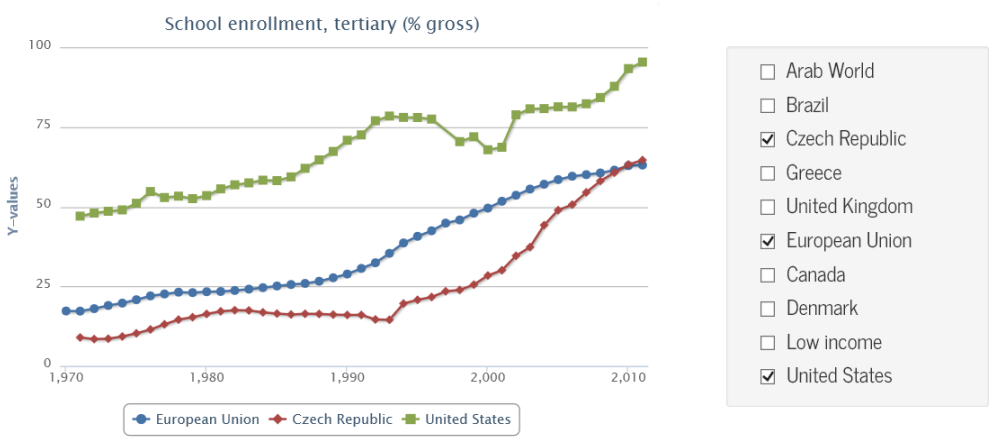

Figure 1: Case study – web application for comparing university enrollment in the world

2.2 Interoperating with JavaScript libraries

To run the sample application on the client-side we use FunScript [4], 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 [17] 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 [24]. We return to this topic in Section 3.3. 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 Service [14], 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 [22]. The F# compiler performs de-sugaring similar to the CPS transformation and interprets keywords such as let! and for using special operations (monadic bind and others). The async identifier determines that we are writing asynchronous workflow [29] 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 the 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 Learning 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 3.2).
- The TypeScript language is unsound due to covariant generics [24]. Thus importing types from TypeScript definitions introduces a potential unsoundness into the F# code (Section 3.3).
- 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 just *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 Integrating with the open world

Type providers [28] are a mechanism for integrating external components into a statically typed programming language. Such components include information sources (such as World Bank), other en-

vironments (here, JavaScript libraries via TypeScript), but type-providers can also be used for limited meta-programming (the Asynchronous parameter can be seen as a form of meta-programming).

3.1 How type providers work and fail

In technical terms, type providers are libraries that are loaded by the compiler (and editor) and are executed at compile-time (or edit-time). A type provider builds information about types it provides and makes those available to the compiler. It is worth noting that this is done lazily, so the type provider does not need to provide types for the entire information space at once.

In terms of programming language theory, type providers can be easily explained anecdotally. They change the starting point for a type-checking of a program as follows:

$$\begin{array}{rl} \emptyset \vdash e : \tau & (\textit{classical, or closed-world}) \\ \pi\ (\text{🌍}) \vdash e : \tau & (\textit{type providers, or open-world}) \end{array}$$

Rather than starting the type-checking with an empty context, we start type-checking with a context containing a *projection* of some information from the world. In this paper, we do not attempt to formalize type providers, so we avoid the question of what, formally, is the world here. However, this analogy is useful for understanding how type providers work and how they can go wrong:

- *Type provider failure.* The projection (a type provider) is implemented in F# and can fail (for example, if the internet is required but unavailable). However, when it succeeds, it generates valid F# code and so unchecked compilation errors cannot happen.
- *Runtime failure.* When the world changes and we *run* the program, we may get a runtime exception. It is expected that the provided code will not fail as long as certain assumptions are satisfied (for example, countries are not removed from a database). But if the world changes and the assumptions no longer hold, the provided code may fail with a runtime exception.
- *Recompilation failure.* When the world changes and we *recompile* the program, the result of the projection can differ and so we may not be able to type-check code that used to type-check. This can be seen as negative aspect, but there is a clear benefit – it means we discover an error that would happen at runtime earlier, that is during recompilation.

The above points are *general* points about type providers. In this paper, we are more interested in the *specific* type providers used in the case study. We look at the World Bank and TypeScript type providers in the following two sections. When discussing type providers, we need to look at how the provider (or the projection π) works and what assumptions it makes about the world. That is, what world changes cause the provided type to change and what world changes cause a runtime exception.

3.2 Integrating with World Bank data

The World Bank provider [20] is a type provider for a specific data source – it has been designed specifically for World Bank data. This contrasts with F# Data type providers for working with CSV, XML and JSON data that take a sample or schema as a parameter and can thus be used with files of any structure.

3.2.1 The World Bank type provider

The projection implemented by the type provider generates a type called Countries that contains countries (and regions) as members. Each member returns a value of type Indicators that is also generated

and contains all indicators as members. Note that the type provider generates just these two types – in particular, it does not generate a new type for each country. This would be an interesting option if we could check what indicators are available for a given country.

The type provider builds the structure of the provided types, but the provided types are *erased* during compilation and replaced with runtime implementation. For the World Bank type provider, the erasure works as follows:

```
⟦ data.``Czech Republic`` ⟧                    = data.GetCountries().GetCountry("CZE")
⟦ cz.``School enrollment, tertiary (% gross)`` ⟧ = cz.AsyncGetIndicator("SE.TER.ENRR")
```

The underlying operations (GetCountry, GetCountries and AsyncGetIndicator) are operations of an underlying runtime library (which is a normal non-provided library). This is a typical pattern – type providers generally produce a light layer on top of a rich runtime library.

An important point here is that the type provider uses the *name* of a country or an indicator for the member name, but at runtime, it uses a *code* of the country or the indicator. This has interesting implications for the safety properties, as discussed in the next section.

3.2.2 Safety properties of the type provider

Now that we know how the World Bank type provider generates the types and the runtime code, we can look at the assumptions it makes about the world and in which cases it encounters one of the failures defined in Section 3.1.

- If you are offline when using the type provider (compiling or editing), the type provider will not be able to obtain the list of countries or indicators. This is an example of *type provider failure*. In case of the World Bank, this is partly prevented by caching of the schema, but internet access is required when using the provider for the first time.
- An interesting case is when a country or indicator is renamed. In that case, compiled code using the World Bank type provider will continue to work (there is no *runtime failure*), because the lookup uses the country or the indicator code. However, recompilation will fail as the member name (which appears in the source code) will be different.
- Finally, if a country or indicator completely disappears, we will get *runtime failure* when running existing compiled code and *recompilation failure*. here, the *recompilation failure* is a useful indicator that the assumptions made by our code have been violated by the data source.

When contrasted with the traditional view of safety in the ML family of languages, the World Bank type provider certainly relaxes the usual safety conditions. However, this is not the case when compared with standard developer practices. In particular, if we wrote `data.GetCountries().GetCountry("CZE")` by hand, the *runtime failure* behaviour would be the same, but we would lose the useful *recompilation failure*. The following section discusses this (and other) alternatives.

3.2.3 Discussion of alternatives

There are two main alternatives that are worth discussing. First, how would the code look if we did not use any type provider. Second, what are alternative designs for the World Bank type provider and how would such alternatives change the safety properties.

Accessing countries without type providers. First, consider how we would write the code to obtain a list of countries without using the World Bank type provider. Assuming we want only the countries and regions in Figure 1, we could specify the country codes and then obtain the country objects:

```
let data = WorldBank.GetDataContext()
let countryCodes = [ "EU", "CZE", "GBP", "USA" ]
let countries = countryCodes |> List.map (data.GetCountries().GetCountry)
```

The motivation for the case study was a data journalism application with a transparent logic that is easy to understand (and perhaps even modify). The type provider certainly achieves these goals better – a country can be added just by looking through an auto-complete list and we do not need to know its code or exact name. However, if we wanted to list all countries in a region, then the World Bank type provider would not be helpful.

As for the safety properties, as discussed earlier, using explicit country codes has the same runtime properties as using the type provider. This could be avoided by listing all countries in a given region (or, according to another rule), but that is solving a different problem.

In summary, the World Bank type provider fits a specific niche that we were exploring in the case study. That is, when we want to create an information analysis that accesses specific data from an information source. With rising popularity of data journalism and Open Government Data initiatives, we argue that this is an important problem domain. Here, we first specify the country codes and then obtain the country objects.

Providing safer data accessors. The *recompilation failure* when a country is removed from the dataset is an indication that code would not behave as expected at runtime and so we find it useful. The problematic case is a *runtime failure* when a country is removed after the program is compiled. One way to avoid this error would be to change the type of the provided members like data.``Czech Republic`` from Country to Country option.

However, that is only shifting the burden of error handling from the library to the user. If the author of the application does not intend to handle errors (for example, by skipping missing data), they would have to write the following:

```
let optCountries =
  [ data.Countries.``European Union``
    data.Countries.``Czech Republic``
    data.Countries.``United Kingdom``
    data.Countries.``United States`` ]
let countries =
  optCountries |> List.map (fun countryOpt →
    match countryOpt with Some c → c | _ → failwith "Country does not exist!")
```

For the purpose of our case study, providing option types would only make the code more complex. However, if we wanted to handle errors more gracefully, then option types would be a better alternative. Note, however, that we can still handle exceptions using try ... with.

Whether using option types is more desirable or not often depends on the use case (how we want to handle errors) and reliability of the data source (countries do not disappear often). This alternative can be supported by adding a static parameter to the type provider (like Asynchronous = true). For example, the CSV type provider from F# Data has a parameter AssumeMissingValues that instructs the type provider to provide option values, even if the sample does not contain them.

Accessing data "as of time". So far, we discussed how to mitigate the problems caused by the fact that the open world changes. Can we make the world *not* change? This is a sensible question for some data sources (although not for the World Bank). For example Freebase [12] supports the notion of "as of time". When calling Freebase with the as_of_time parameter, the API returns data and meta-data that was present at the specified date.

This would be a desirable option for our case study – if we want to illustrate point made by an article, the accompanying application could use data available at the time when the article was written. However, this feature relies on the ability of the data source (not all services have this), or on the possibility to create local snapshots (difficult for large data sources).

3.3 Integrating with JavaScript libraries

Type providers are often described as a technique that simplifies data access. The TypeScript type provider shows that there is a broader range of uses, including integration between different programming languages. Another example of type provider for language integration is the R provider [3] which imports packages and functions of the statistical environment R.

3.3.1 The TypeScript type provider

The projection implemented by the TypeScript type provider imports interface definitions in the `".d.ts"` format specified by TypeScript [17]. This is a format of type annotations for JavaScript libraries. The following example is an excerpt from the `"jquery.d.ts"` file that describes the jQuery value and the attr method used in our example:

```
declare var jQuery : JQueryStatic;

interface JQueryStatic {
  (selector : string, context? : any) : JQuery;
}

interface JQuery {
  attr(attributeName : string) : string;
  attr(attributeName : string, value : any) : JQuery;
}
```

The example demonstrates a number of typical problems that arise when using type providers for language interoperation. Not all TypeScript language constructs have direct equivalent in F# and the TypeScript type provider needs to map them to other constructs that are available:

- The interface file defines jQuery as a global variable. Although F# supports global bindings, those cannot be provided by a type provider and so the global variable is exposed as a static member of the imported type, so we use it via j.jQuery.
- The interface JQueryStatic specifies that the jQuery object is callable (other members are omitted). This denotes that the corresponding JavaScript object is a callable function with other members. F# does not allow "calling an object" and so this is mapped to a method j.jQuery.Invoke.

It is also worth noting that the attr member is overloaded and uses an optional parameter. However, both of these features are available in F# and can be directly mapped. Now that we covered how the type

structure is generated, we also need to show what code is generated when the types are erased:

```
⟦ j.jQuery.Invoke(command) ⟧ =
    let jq = Emit.PropertyGetImpl(true, "jQuery", [| |])
    Emit.CallImpl(false, "", [| jq; command |] )

⟦ jQuery("<input>").attr("type","checkbox") ⟧ =
    Emit.CallImpl(false,"attr",[| jQuery("<input>"); "type"; "checkbox" |] )
```

An important point here is that the above provided code is never actually *executed*. As discussed in Section 4.1, it is translated to JavaScript (which then runs in the web browser). So, the above code can be seen more as instructions for the translator. The mapping is quite simple – member calls are translated to Emit.CallImpl (empty name denotes that the object itself is called) and property getters are translated to Emit.PropertyGetImpl. In both cases the first argument denotes whether the call is static and the last argument is an array of arguments.

3.3.2 Safety of cross-language type providers

As mentioned earlier, TypeScript does not have the type soundness property [24], because of covariant generics. This illustrates two points. First, this is a property of *running* TypeScript code, which is not the case here – we use TypeScript only as a source of annotations for JavaScript libraries. There is no guarantee that the library will actually adhere to the specification, which is an issue that we return to when discussing translation to JavaScript (Section 4.1).

The second point is that we can not import the typing rules associated with TypeScript. The type provider mechanism can only provide F# types that then behave according to the F# typing rules. This means that we can import generics[4], but this will not automatically allow using them covariantly. In some cases, the F# type system is simply more strict – and in that case, users have to use unsafe operations, such as unbox⟨bool⟩ in the render function. We return to the topic of unsafe operations in Section 4.1.

The above discussion highlights a broader point about using type providers for language interoperability. Given any language, its type system be weaker or stricter in some ways. Both cases make mapping difficult:

- First, consider a weaker system with less type information. The type provider may need to map more types to object (or similar general type). However, this will make the provided operations hard to use (because the source language is more flexible than F#). An alternative is to require explicit annotations (like `d.ts` files for JavaScript).
- Second, if the imported language has a more precise type system than F#, the type provider has to drop some of the information. This can be done safely for types in contravariant positions. In the unsafe case, the runtime needs to perform dynamic checks (Section 5).

In case of JavaScript, we encounter both cases. In many cases, the return type is not statically known and is exposed as any, which we then map to F# object (and the developer has to use unbox). However, TypeScript also supports limited form of dependent typing (overloading on constants [30]). This cannot be expressed in the F# type system and the developer has to do more work (for example, choose the right overload or provide an explicit type annotation).

[4]Due to technical limitations, this is not currently allowed by F# type providers, but there are no fundamental reasons for this.

An interesting example of type provider for a language with a weaker type system is the R provider [3]. It solves the mismatch by using runtime reflection to discover R functions and their declared parameters. However, R provider also generates additional function overloads that let the user specify additional parameters (not explicitly declared in the signature). To make such interoperability easier, we proposed additional escape mechanism [18] that provide usable syntax when static type information cannot be fully recovered.

3.3.3 Discussion of alternatives

There are two options for calling JavaScript libraries such as jQuery. We can use annotations – written either in F# or imported into F#, or we can let the user write inline JavaScript. A different approach is to discourage the use of existing JavaScript libraries as discussed in Section 4.2.

Embedding inline JavaScript. To make the interoperability mechanism general, the host language (here, F#) should not know about the syntax of the embedded language (here, JavaScript). This means that the JavaScript code can only be embedded as strings. FunScript provides this option and it is used for accessing various primitive JavaScript operations. For example, the following is one way to define a function that converts a value to a number using JavaScript semantics:

```
[⟨JSEmit("return {0}*1.0;")⟩]
let number (_ : obj) : float = failwith "JavaScript stub should not be called."
```

The example uses .NET attributes (on the first line), which are meta-data attached to a function. When FunScript finds a call to such annotated function, it replaces the call with the specified JavaScript. The string `{0}` is a placeholder for the argument. Note that the body of the function is never actually executed and so we use a placeholder.

The approach provides no guarantees about the inline JavaScript code. At compile-time, it cannot even check that the JavaScript code is syntactically correct. This could be improved using a parameterized type provider such as `JSCode⟨"return {0}*1.0;"⟩`, which would check that the static parameter is syntactically correct JavaScript code. So, type providers could be used to make this approach safer.

Other ways of writing annotations. Many compilers to JavaScript provide a way for writing type signatures for JavaScript libraries in the host language. For example, the following snippet shows a declaration of the jQuery `attr` method from a mapping for js_of_ocaml [31] (definitions in SMLtoJs [10] are similar, but simpler):

```
class type jQuery = object
  method attr : js_string t → js_string t optdef meth
  method attr_set : js_string t → js_string t → jQuery t meth
end
```

The js_of_ocaml project uses the OCaml object model for calling JavaScript libraries. Compared with FunScript, it is more explicit in importing JavaScript – the `meth` type denotes a JavaScript method; `optdef` specifies that the result may be undefined and `js_string` denotes a JavaScript string (which is distinct from OCaml strings). Also, note that the second method is called `attr_set`. This is a simple naming trick – OCaml does not support overloading and js_of_ocaml simply ignores anything after underscore.

Using a type provider is similar to writing a code generator that turns TypeScript `d.ts` files into the above OCaml annotation. The main difference is that type providers provide a smoother integration. Type providers do not produce any artefacts (generated files) and can import entire repositories lazily.

3.4 Relativized type safety

Type providers for information sources, such as the World Bank provider, weaken the usual notion of type safety. In general, they require that certain assumptions about the world do not change between the state of the world at compile-time and state of the world at run-time. The TypeScript provider is different and it is more interesting to consider runtime behaviour as discussed in Section 5.

For type providers that provide access to external information sources, we can formulate a *relativized* notion of type safety. A full formalization is beyond the scope of this paper, but the following provides an outline of such a theorem.

Theorem (Relativized type safety)**.** *Assume* $\pi_{\mathsf{WorldBank}}$ *is the mapping implemented by the World Bank type provider,* w, w_0 *are models of a (read-only) world that can be seen as functions defined on country-indicator pairs and* $\langle e, w\rangle \to e'$ *is a one-step reduction on expressions that has access to the World Bank data modelled by* w*. Then we define:*

- *Multi-step reduction* $\langle e, w\rangle \to^* e'$ *if either* $e = e'$ *or* $\langle e, w\rangle \to e''$ *and* $\langle e'', w\rangle \to^* e'$*.*
- *Functions* $\mathsf{countries}(e)$ *and* $\mathsf{indicators}(e)$ *that return sets of country names and indicator names that appear in an expression* e*.*

Then the following relativized type safety *implication holds:*

$$
\begin{array}{ll}
(\ \pi_{\mathsf{WorldBank}}\ (w_0) \vdash e : \tau & \text{If } e \text{ is well-typed using a compile-time world } w_0 \\
\quad \wedge \langle e, w\rangle \to^* e' & \ldots \text{and the expression reduces in one or more steps} \\
\quad \wedge \ (\ \forall c \in \mathsf{countries}(e), & \ldots \text{and the run-time world model } w \text{ contains all} \\
\qquad \forall i \in \mathsf{indicators}(e).\langle c, i\rangle \in \mathit{dom}(w))) & \quad \text{country-indicator pairs that may be accessed,} \\
\Rightarrow \ \mathsf{value}(e') \ \vee \ (\exists e''.\langle e', w\rangle \to e'') & \text{then the result is either a value or can further reduce.}
\end{array}
$$

The theorem has the usual structure of a type safety theorem. The reduction is defined on pairs consisting of expression and a world w. The result of the reduction is only an expression and so it cannot modify the world. Even though the above is only a brief sketch, it demonstrates two important points about relativized type safety that are relevant to many type providers.

First, we distinguish between the state of the world w_0 that is used at compile-time and the state w used at run-time. If these were the same, the additional assumption would always hold. Second, the additional assumptions states that all countries and indicators that *may be accessed* need to be available. This is stricter than necessary (akin to effect systems), but it does not specify that the run-time world must contains *all* provided countries and indicators.

4 Compiling to JavaScript

JavaScript has become the *lingua franca* of the web and an increasing number of programming languages provide a way of compiling to JavaScript. In F#, the first project was F# WebTools [21] in 2007. A more recent and complex framework called Websharper [2] is available with full commercial support. In this paper, we use FunScript which is a lightweight library focused just on translating F# to JavaScript.

In this section, we discuss the options available when targeting JavaScript. The compilation to JavaScript can be implemented as compiler back-end or as a library based on meta-programming (Section 4.1), there are different approaches to base libraries (Section 4.2), asynchronous computing (Section 4.3) and the semantic mismatch between the host language and JavaScript (Section 4.4).

4.1 Lightweight meta-programming with quotations

The FunScript library is based on F# quotations [27]. This means that the code is compiled as ordinary F#, but the compiler also stores marked blocks of code as data. In FunScript, we use a launcher that, when the program runs, analyses its quotations and produces a JavaScript file (so, the workflow is to compile the code as usual and then run it to produce JavaScript).

In FunScript, we typically need to translate the entire source file. To allow this, we instruct the compiler to store code as data for the whole module using the ReflectedDefinition attribute. This is done at the beginning of the file (which we omitted in Section 2):

```
[⟨ReflectedDefinition⟩]
module Program

open FunScript
open FunScript.TypeScript
open FSharp.Data
```

When marked with ReflectedDefinition, the F# compiler stores the body of all functions and methods in the marked module as *quotations* that can be retrieved at run-time. Note that this does not store the AST for the entire file – for example, information about declared types needs to be obtained separately using the .NET reflection mechanism.

The quotation mechanism has other uses. In the following, we use an explicit quotation $\langle @ \ldots @ \rangle$, which returns the wrapped code as data. This can be used, for example, to compile a matrix calculation to run on a GPU [26]:

```
⟨@ fun (input : Matrix⟨float32⟩) →
      let sum = (shift input −1 0) + input + (shift input +1 0)
      sum / 3.0f @⟩
```

Compared to writing a full compiler back-end, the lightweight approach to meta-programming makes it easy to write a translator from F# to other languages. This has been used for compiling to SQL queries, compiling F# code to CUDA, for Freebase queries [20] and other applications.

Using the lightweight meta-programming approach in FunScript would not be appropriate if we wanted to compile arbitrary existing F# source code to JavaScript without any modifications. For that, using a compiler back-end is a better choice. However, for the task solved in our case study (compiling newly written F# code), the approach works well. A related question is providing mappings for standard (in our case, F# and .NET) libraries used by the code. We discuss this in the next section.

4.2 Accessing F# and JavaScript libraries

The F# code in our case study uses a number of libraries. This includes the F# core library (for example, the List.map function), standard .NET libraries (iteration using the for loop uses .NET IEnumerable interface) and JavaScript libraries. Accessing these libraries in FunScript follows a similar approach to F# WebTools [21].

The range of possible approaches to libraries when translating to JavaScript has two extreme cases. We can attempt to port all libraries of our source language (F#, Haskell or even .NET), or we can ignore most standard libraries and instead provide access to as many JavaScript libraries as possible. FunScript stands in the middle – it provides access to *some* F# and .NET libraries, but most advanced functionality is accessed through JavaScript libraries.

Mapping standard .NET and F# libraries. The F# ecosystem relies on .NET libraries. This means that taking an F# library and translating it to JavaScript without any modification is not a viable approach as we would have to be able to also translate any (compiled) .NET library. This is where F# differs, for example, from Haskell which has a closed ecosystem and the ghcjs project [16] is thus capable of translating many standard libraries.

FunScript implements (explicit) mappings for standard F# and .NET libraries that are useful in JavaScript (collections, string manipulation, date and other). Those are either reimplemented in F# with the ReflectedDefinition attribute, or mapped to standard JavaScript library.

An interesting aspect of the case study is that we use F# Data [20] type providers, which is a standard F# library that has not been built specifically for FunScript. This illustrates how type providers and quotations interact.

When we write code using type providers that provide erased types (both World Bank and TypeScript), then the erasure happens before a quotation is captured. The following listing shows an example. When we write code with explicit (or implicit) quotation containing a provided method (1), the actual quotation (2) contains just calls to the underlying runtime library:

```
⟨@ data.``Czech Republic`` @⟩                          (1)
⟨@ data.GetCountries().GetCountry("CZE") @⟩             (2)
```

Thus, the FunScript component only needs to provide a client-side implementation of the underlying operations GetCountries, GetCountry and AsyncGetIndicator. This is done in the same way as mappings for standard .NET libraries. Under the cover, AsyncGetIndicator invokes an AJAX request to the World Bank API. We provide more details about asynchronous execution in Section 4.3.

Accessing standard JavaScript libraries. As discussed earlier, client-side programs can either rely more heavily on JavaScript libraries, or they can reimplement most of the core functionality. The latter approach is easier when all code is written for the same ecosystem. This is done, for example, by ghcjs [16]. This technique can be very effective when writing code that will be a part of a larger JavaScript application and implements, for example, a core computation. This can then be compiled to run efficiently using asm.js [8].

The case study presented in this paper focuses more on the plumbing code that connects multiple components – F# type providers for data access and JavaScript libraries for visualization. Thus, a smooth integration between F# and JavaScript is necessary.

An important problem with accessing JavaScript libraries is deciding which types should be mapped to ordinary F# types and which types should be treated as opaque JavaScript types that can only be manipulated by JavaScript operations and operators. For example, consider the following excerpt from the code in Section 2.3:

```
let o = h.HighchartsOptions()
o.chart ← h.HighchartsChartOptions(renderTo = "plc")
o.title ← h.HighchartsTitleOptions(text = head)
o.series ← [| |]
(...)

opts.series.push(h.HighchartsSeriesOptions(data, name))
(...)
```

In ordinary F# code, we would not initialize o.series to an empty array before accessing it and we would not expect to use a push method to append an element – .NET arrays are mutable, but not resizable.

Here, the first is necessary. When created, the series property of HighchartsOptions is undefined. The push operation is added (as an extension method) to standard .NET array, but it cannot be implemented for ordinary F# code. The following list briefly summarizes the options for mapping JavaScript libraries to F# and the choices made in FunScript:

- Numeric types including int and float are mapped to JavaScript numbers. However, the floating-point arithmetic in JavaScript differs from the one in F# and so this changes the semantics. Similarly, compilation of integer operators requires additional work (for example, $1/2$ is not an integer division in JavaScript).
- FunScript maps F# arrays to JavaScript arrays. We can define extension methods such as push that make sense for JavaScript arrays. An alternative is to use a separate type – for example js_of_ocaml uses a separate js_array type. A nice thing about the FunScript approach is that it lets us use standard F# modules and functions such as Array.map.
- Due to its .NET heritage, there are many types in F# that have null as a valid value. This means that there is a reasonable precedent for allowing undefined values on types imported from JavaScript (but not for types defined in F#). In contrast, js_of_ocaml is more explicit – for example the attr operation is marked as returning optdef (potentially undefined) js_string value whereas FunScript uses just .NET string (which may be null).

In summary, the approach used by F# is to reuse as much of F# and .NET as possible. This makes writing and reusing code easier and lets developers use existing familiar libraries and idioms. However, it means that the semantics of F# running as JavaScript is not the same as the semantics of F# running as compiled code. We return to this topic in Section 4.4. Before that, the following section gives one more example where similar approach is used.

4.3 Client-side asynchronous computations

In F#, asynchronous workflows [29] serve two purposes. First, they provide a way for running multiple tasks in parallel. Second, they make it possible to write long-running non-blocking code without the use of explicit callbacks. In our case study, we only use the latter aspect. When loading data for selected countries in a loop, we fetch data asynchronously (using an AJAX call). Under the cover, JavaScript triggers a callback, which then resumes the asynchronous workflow.

Although the recent HTML5 standard makes it possible to write multi-threaded JavaScript using Web Workers, the FunScript implementation of async $\{\ldots\}$ does not do that. Instead, it uses a simple cooperative model based on continuations (which is similar to how Links [6] compilation works).

In ordinary F#, asynchronous workflows can be started in a number of ways: Async.RunSynchronously starts the work and blocks until it completes; Async.Start starts the work in the background and Async.StartImmediate starts the work on the current thread and runs callbacks in the same synchronization context (essentially on the same thread). However, only the last one can be mapped to JavaScript (FunScript translator will fail if the other two are used).

Our approach is again to reuse standard F# constructs, but interpret them in a way that fits with the new execution environment. The alternative would be to define a separate computation type such as js_async $\{\ldots\}$. So, developers can use existing constructs, but need to be aware that their code runs in a different environment.

4.4 Relativized semantics

When compiling any language to JavaScript, we can treat JavaScript as a low-level runtime (and use the asm.js [8] subset for efficiency), or we can use it as higher-level language and map many source language constructs to corresponding JavaScript constructs. In this case study, we used the latter. This enables interop with JavaScript, but it makes it hard (if at all possible) to preserve the original F# semantics in all cases.

This is perhaps a controversial approach, but it fits well with F#, because it is similar to how F# handles interop with .NET. When you do not use .NET objects, you do not need type annotations and you do not have to handle `null` values. If you use .NET, you have to accept those at some level. More generally, we could call this property *relativized semantics*:

> *When you use the ML subset of F#, the program will behave in the same way regardless of the environment and you are guaranteed the usual ML properties. In other environments, the properties are not guaranteed.*

Making the notion of *relativized semantics* formally precise is outside of the scope of this paper. An important aspect is specifying the boundary (see Section 5). The problem is similar to writing monadic computations – each monad is different (and has different properties), but there are certain laws that always hold. For full cross-compilation, we need a similar set of laws, but for all ML language constructs.

5 Related and further work

Related work. There are a number of projects that solve similar problems to the ones described in this paper. In the F# ecosystem, the first project compiling to JavaScript was F# WebTools [21], which also supported asynchronous computations (although using a separate computation type) and interop with JavaScript. More recently, WebSharper [9, 2] is a more complete framework for web development that also provides composable abstractions for building forms (based on formlets [7]) and entire applications.

Other statically-typed functional languages that have some way of running as JavaScript include OCaml [31], SML [10] and Haskell [16]. Compared to our work, all three are stricter in preserving the semantics of the source language, but do not provide as smooth JavaScript integration.

Further work. The programming model used in our case study combines dynamically typed components (JavaScript libraries) and statically typed components (code written in F#). This suggests a relation with the work on gradual typing [25] and blame tracking [32]. However, in our work, the distinction between the statically and dynamically typed parts (and the boundary) is less explicit. However, it would be interesting to see if gradual typing and blame can be used to formally define our informal notion of *relativized semantics* introduced in Section 4.4.

6 Conclusions

The key idea of this paper is that writing modern applications for the web requires us to reconsider many assumptions that functional language designers take for granted. The novelty of this paper is not in any single technology it presents, but in the combination it shows. For this reason, we used the form of a case study, using a significant example to guide our discussion.

There are two main assumptions that we reconsider. The first assumption is that programs operate in a closed world. Instead, modern application access a range of external services and information-sources. In our case study, these are accessed using type providers. The second assumption is that we can fix the runtime semantics. This becomes difficult when the same code is compiled for diverse execution environments such as .NET, JavaScript or even CUDA.

If there is one thing that the reader should remember from this case study, it is the idea about F# saying that "*when you use it as ML, it behaves as ML*". That is, when you use the ML subset of F# and do not access external services, information or .NET and JavaScript libraries, you still get all the good properties of ML. However, when you access external information and run your code as JavaScript, you get a weaker notion of safety that we called *relativized type safety* and a weaker runtime guarantees that we called *relativized semantics*. Nevertheless, the case study shows that these are enough to let us benefit from the functional-first statically-typed programming paradigm in the age of web.

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 IFL*, IFL'10, pp. 203–216.

[3] BlueMountain Capital and Contributors (2015): *F# R Type Provider*. Available at `http://bluemountaincapital.github.io/FSharpRProvider`.

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

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

[6] Ezra Cooper, Sam Lindley, Philip Wadler & Jeremy Yallop (2007): *Links: Web Programming Without Tiers*. In: *Proceedings of FMCO*, FMCO'06, pp. 266–296. Available at `http://dl.acm.org/citation.cfm?id=1777707.1777724`.

[7] Ezra Cooper, Sam Lindley, Philip Wadler & Jeremy Yallop (2008): *The Essence of Form Abstraction*. In: *Proceedings of APLAS*, APLAS '08, pp. 205–220, doi:10.1007/978-3-540-89330-1_15.

[8] Alon Zakai David Herman, Luke Wagner (2014): *asm.js – Working Draft – 18 August 2014*. Available at `http://asmjs.org/spec/latest/`.

[9] Loïc Denuzière, Ernesto Rodriguez & Adam Granicz (2014): *Piglets to the Rescue: Declarative User Interface Specification with Pluggable View Models*. In: *Proceedings of IFL*, IFL '13, ACM, pp. 105:105–105:115, doi:10.1145/2620678.2620689.

[10] Martin Elsman (2011): *SMLtoJs: Hosting a Standard ML Compiler in a Web Browser*. In: *Proceedings of PLASTIC*, PLASTIC '11, ACM, pp. 39–48, doi:10.1145/2093328.2093336.

[11] Sigbjorn Finne, Daan Leijen, Erik Meijer & Simon Peyton Jones (1999): *Calling Hell from Heaven and Heaven from Hell*. In: *Proceedings of ICFP*, ICFP '99, ACM, pp. 114–125, doi:10.1145/317636.317790.

[12] Google (2015): *Freebase: A community-curated database of well-known people, places, and things*. Available at `http://www.freebase.com`.

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

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

[15] 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.

[16] Hamish Mackenzie Luite Stegeman & Contributors (2015): *ghcjs: Project homepage*. Available at `https://github.com/ghcjs`.

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

[18] Tomas Petricek (2014): *F# Language: Allow "params" dictionaries as method arguments*. Available at `https://fslang.uservoice.com/forums/245727/suggestions/5975840`.

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

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

[21] Tomas Petricek & Don Syme (2007): *F# Web Tools: Rich client/server web applications in F#*. Unpublished draft, submitted to ML Workshop 2007. Available at `http://tomasp.net/academic/articles/fswebtools`.

[22] Tomas Petricek & Don Syme (2014): *The F# Computation Expression Zoo*. In: *Proceedings of PADL*, PADL 2014, Springer-Verlag New York, Inc., pp. 33–48, doi:10.1007/978-3-319-04132-2_3.

[23] Simon L. Peyton Jones & Philip Wadler (1993): *Imperative Functional Programming*. In: *Proceedings of POPL*, POPL '93, ACM, pp. 71–84, doi:10.1145/158511.158524.

[24] 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`.

[25] Jeremy G Siek & Walid Taha (2006): *Gradual typing for functional languages*. In: *Scheme and Functional Programming Workshop*, 6, pp. 81–92.

[26] Satnam Singh (2010): *Declarative Data-parallel Programming with the Accelerator System*. In: *Proceedings of DAMP*, DAMP '10, ACM, pp. 1–2, doi:10.1145/1708046.1708048.

[27] Don Syme (2006): *Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution*. In: *Proceedings of ML Workshop*, ML '06, ACM, pp. 43–54, doi:10.1145/1159876.1159884.

[28] Don Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu & T. Petricek (2012): *Strongly-typed language support for internet-scale information sources*. Technical Report MSR-TR-2012-101, Microsoft Research. Available at `http://research.microsoft.com/apps/pubs/?id=173076`.

[29] Don Syme, Tomas Petricek & Dmitry Lomov (2011): *The F# Asynchronous Programming Model*. In: *Proceedings of PADL*, PADL'11, pp. 175–189.

[30] Jonathan Turner (2013): *Announcing TypeScript 0.9*. Available at `http://blogs.msdn.com/b/typescript/archive/2013/06/18/announcing-typescript-0-9.aspx`.

[31] Jérôme Vouillon & Vincent Balat (2013): *From bytecode to javascript: the js_of_ocaml compiler*. *Software: Practice and Experience*.

[32] Philip Wadler & Robert Bruce Findler (2009): *Well-Typed Programs Can'T Be Blamed*. In: *Proceedings of ESOP*, ESOP '09, pp. 1–16, doi:10.1007/978-3-642-00590-9_1.