F# Data: Making structured data first-class citizens

Tomas Petricek
University of Cambridge
tomas@tomasp.net

Gustavo Guerra
Microsoft Corporation, London
gustavo@codebeside.org

Don Syme
Microsoft Research, Cambridge
dsyme@microsoft.com

Abstract

Most modern applications interact with external services and access data in formats such as XML, JSON and CSV. Static type systems do not understand external data sources and only make accessing data more cumbersome. Should we give up and leave the messy world of external data to dynamic typing and runtime checks? Of course, not!

We show how to integrate external data sources into the F# type system. As most real-world data do not come with an explicit schema, we develop a type inference algorithm that infers a type from representative sample documents and integrate it into the F# type system using type providers.

Our library significantly reduces the amount of code that developers need to write when accessing data. It also provides additional safety guarantees, arguably, as much as possible if we abandon the closed world assumption.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords F#, Type Providers, Inference, JSON, XML

1. Introduction

Applications for taking notes, searching train schedules or finding tomorrow's weather all communicate with one or more services over the network and present the aggregated data. Increasing number of such services provide REST-based end-points that return data as CSV, XML or JSON. Despite numerous schematization efforts, most services do not come with an explicit schema. At best, the documentation provides sample responses for typical requests.

For example, OpenWeatherMap provides an end-point for getting current weather for a given city[1]. The documentation describes the URL parameters and shows one sample JSON to illustrate the typical response structure. Using a standard library for working with JSON and HTTP, we might call the service and read the temperature as follows:

```
let doc = Http.Request("http://weather.org/?q=NYC")
match JsonValue.Parse(doc) with
| Record(root) →
  match Map.find "main" root with
  | Record(main) →
    match Map.find "temp" main with
    | Number(num) → printfn "Nice %f degrees!" num
    | _ → failwith "Incorrect format"
  | _ → failwith "Incorrect format"
| _ → failwith "Incorrect format"
```

The code assumes that the response has a particular format described in the documentation. The root node must be a record with a `"main"` field, which has to be another record containing a numerical `"temp"` field. When the format is incorrect, the data access simply fails with an exception.

Using the JSON type provider from F# Data, we can write code with exactly the same functionality in two lines:

```
type W = JsonProvider⟨"http://weather.org/?q=NYC"⟩
printfn "Nice %f degrees!" (W.GetSample().Main.Temp)
```

Here, `JsonProvider⟨"..."⟩` invokes a type provider at compile-time with the URL as a sample. The type provider infers the structure of the response and provides a type with a `GetSample` method that returns a parsed JSON with nested properties `Main.Temp`, returning the temperature as a number.

The rest of the paper describes the mechanism and discusses its safety properties. The key novel contributions are:

- We present F# Data type providers for XML, CSV and JSON (§2) and practical aspects of their implementation that contributed to their industrial adoption (§6).
- We describe a predictable type inference algorithm for structured data formats that underlies the type providers (§3) based on a *common supertype* relation.
- We give a formal model (§4) and use it to prove *relativized type safety* for the type providers (§5). This adapts the ML-style type safety for the context of the web.

[1] See "Current weather data": http://openweathermap.org/current

2. Structural type providers

We start with an informal overview that shows how F# Data type providers simplify working with JSON, XML and CSV. We also introduce the necessary aspects of F# 3.0 type providers along the way. The examples in this section illustrate a number of key properties of our type inference algorithm:

- Our mechanism is robust and predictable. This is important as the user directly works with the inferred types and should understand why a specific type was inferred from a given sample[2].
- Our inference mechanism prefers records over unions. This better supports developer tooling – most F# editors provide code completion hints on "." and so types with properties (records) are easier to use than types that require pattern matching.
- Finally, we handle a number of practical concerns that may appear overly complicated, but are important in the real world. This includes support for different numerical types, null values and missing data and also different ways of representing Booleans in CSV files.

2.1 Working with JSON documents

The JSON format used in the example in Section 1 is a popular format for data exchange on the web based on data structures used in JavaScript. The following is the definition of the JsonValue type used earlier to represent parsed JSON:

```
type JsonValue =
  | Null
  | Number of decimal
  | String of string
  | Boolean of bool
  | Record of Map⟨string, JsonValue⟩
  | Array of JsonValue[]
```

The OpenWeatherMap example in the introduction used only a (nested) record containing a numerical value. To demonstrate other aspects of the JSON type provider, we look at a more complex example that also involves null value and an array:

```
[ { "name": "Jan", "age": 25 },
  { "name": "Alexander", "age": 3.5 },
  { "name": "Tomas" } ]
```

Say we want to print the names of people in the list with an age if it is available. As before, the standard approach would be to pattern match on the parsed JsonValue. The code would check that the top-level node is a Array, iterate over the elements checking that each is a Record with certain properties and throw an exception or skip values in incorrect format. The standard approach can be made nicer by defining helper functions. However, we still need to specify names of fields as strings, which is error prone and can not be statically checked.

Assuming the people.json file contains the above example and data is a string value that contains another data set in the same format, we can print names and ages as follows:

```
type People = JsonProvider⟨"people.json"⟩

let items = People.Parse(data)
for item in items do
  printf "%s " item.Name
  Option.iter (printf "(%f)") item.Age
```

The code achieves the same simplicity as when using dynamically typed languages, but it is statically type-checked. In contrast to the earlier example, we now use a local file people.json as a sample for the type inference, but then processes data from another source.

Type providers. The notation JsonProvider⟨"people.json"⟩ on the first line passes a *static parameter* to the type provider. Static parameters are resolved at compile-time, so the file name has to be a constant. The provider analyzes the sample and generates a type that we name People. In editors that use the F# Compiler Service, the type provider is also executed at development-time and so the same provided types are used in code completion.

The JsonProvider uses a type inference algorithm (Section 3) and infers the following types from the sample:

```
type Entity =
  member Name : string
  member Age : option⟨decimal⟩

type People =
  member GetSample : unit → Entity[]
  member Parse : string → Entity[]
```

The type Entity represents the person. The field Name is available for all sample values and is inferred as string. The field Age is marked as optional, because the value is missing in one sample. The two sample ages are an integer 25 and a decimal 3.5 and so the common inferred type is decimal.

The type People provides two methods for reading data. GetSample returns the sample used for the inference and Parse parses a JSON string containing data in the same format as the sample. Since the sample JSON is a collection of records, both methods return an array of Entity values.

Error handling. In addition to the structure of the types, the type provider also specifies what code should be executed at run-time in place of item.Name and other operations. This is done through a *type erasure* mechanism demonstrated in Section 2.3. In this example, the runtime behaviour is the same as in the hand-written sample in Section 1 – a member access throws an exception if the document does not have the expected format.

[2] In particular, we do not use probabilistic methods where adding additional sample could change the shape of the type.

Informally, the safety property discussed in Section 5 states that if the inputs are subtypes of some of the provided samples (*i.e.* the samples are representative), then no exceptions will occur. In other words, we cannot avoid all failures, but we can prevent some. If the OpenWeatherMap changes the response format, the sample in Section 1 will not re-compile and the user knows that the code needs to be changed. What this means for the traditional ML type safety is discussed in Section 5.1.

The role of records. The sample code is easy to write thanks to the fact that most F# editors provide code completion when "." is typed. The developer does not need to look at the sample JSON file to see what fields are available for a person. To support this scenario, our inference algorithm prefers records (we also treat XML elements and CSV rows as records).

In the above example, this is demonstrated by the fact that Age is marked as optional. An alternative is to provide two different record types (one with Name and other with Name and Age), but this would complicate the processing code.

2.2 Processing XML data, alternative

In XML, documents are formed by (possibly) nested elements with attributes. We can view elements as records (unlike in JSON, the records are named) with a field for each attribute and an additional special field for the nested contents (which may be a collection of elements).

As an example, consider a simple extensible document format where a root element `<doc>` can contain a number of document element, one of which is `<heading>` representing headings:

```
<doc>
  <heading>Working with JSON</heading>
  <p>Type providers make this easy.</p>
  <heading>Working with XML</heading>
  <p>Processing XML is as easy as JSON.</p>
  <image source="xml.png" />
</doc>
```

The F# Data library has been primarilly designed to allow easy reading of data. For example, say we want to print all headings that appear in the document. The sample shows a part of the possible document structure (in particular the `<heading>` element we want to read), but it does not show all possible elements (say, `<table>`). Assuming the above document is sample.doc, we can write:

```
type Document = XmlProvider⟨"sample.doc"⟩
let root = Document.Load("http://.../another.doc")
for elem in root.Doc do
  match elem.Heading with
  | Some text → printf " - %s " text
  | None → ()
```

The example iterates over a collection of elements elem returned by root.Doc. Our system provides a typed access to elements known from the sample document and so we can write elem.Heading.

Open world and variant types. In the above example, The system does not infer a union type representing a choice between heading, paragraph and image. By its nature, the XML format is extensible and the sample cannot include all possible examples[3]. The type of elem is thus a *variant* type annotated with the three statically known possibilities (heading, paragraph, image). The variant is mapped to the following F# type:

```
type Element =
  member Heading : option⟨string⟩
  member Paragraph : option⟨string⟩
  member Image : option⟨Image⟩
```

The type provides a nice access to the elements known statically from the sample, but it encodes the *open-world assumption* – the user also needs to handle the case when the value is none of the statically known elements.

One perhaps surprising consequence of the use of variant types is that the above code will work correctly on document that contains elements other than those in the sample document. This would arguably not be desirable design for data types in a programming language, but it is extremely useful when reading data from external data sources. It is also worth noting that our type inference attempts to minimze the number of variant types that appear in the resulting type – a variant is not used if there is another option.

2.3 Reading CSV files

In the CSV file format, the structure is a collection of rows (records) consisting of fields (with names specified by the first row). The inference needs to infer the types of fields. For example:

```
Ozone, Temp, Date,       Autofilled
41,    67,   2012-05-01, 0
36.3,  72,   2012-05-02, 1
12.1,  74,   3 kveten,   0
17.5,  #N/A, 2012-05-04, 0
```

One difference between JSON and CSV formats is that in CSV, the literals have no data types. In JSON, strings are "quoted" and Booleans are true and false. This is not the case for CSV and so we need to infer not just the structure, but also the primitive types.

Assuming the sample is saved as airdata.csv, the following snippet prints all rows from another file that were not

[3] Even when the document structure is defined using XML Schema, documents may contain elements prefixed with other namespaces.

autofilled:

```
type AirCsv = CsvProvider<"airdata.csv">
let air = AirCsv.Parse(data)
for row in air.Rows do
  if not row.Autofilled then
    printf "%s: %d" row.Date row.Ozone
```

The type of the record (row) is, again, inferred from the sample. The Date column uses mixed formats and is inferred as string (although we support many date formats and "May 3" would work). More interestingly, we also infer Autofilled as Boolean, because the sample contains only 0 and 1 values and using those for Booleans is a common CSV convention. Finally, the fields Ozone and Temp have types decimal and option<int>.

2.4 Summary

The previous three sections demonstrated the F# Data type providers for JSON, CSV and XML. A reader interested in more examples is invited to look at documentation on the F# Data web site[4].

It should be noted that we did not attempt to present the library using well-structured ideal samples, but instead used data sets that demonstrate typical issues that are frequent in real world inputs (missing data, inconsistent encoding of Booleans and heterogeneous structures).

From looking at just the previous three examples, these may appear arbitrary, but our experience suggests that they are the most common issues. The following JSON response with government debt information returned by the World Bank API demonstrates all three problems:

```
[ { "page": 1, "pages": 5 },
  [ { "indicator": "GC.DOD.TOTL.GD.ZS",
      "date": "2012",
      "value": null },
    { "indicator": "GC.DOD.TOTL.GD.ZS",
      "date": "2010",
      "value": "35.1422970266502" } ] ]
```

First of all, the top-level element is a collection containing two values of different kind. The first is a record with metadata about the current page and the second is an array with data. The JSON type provider infers this as heterogeneous collection with properties Record for the former and Array for the latter. Second, the value is null for some records. Third, numbers can be represented in JSON as numeric literals (without quotes), but here, they are returned as string literals instead[5].

[4] Available at http://fsharp.github.io/FSharp.Data/

[5] This is often used to avoid non-standard numerical types in JavaScript.

3. Structural type inference

Our type inference algorithm for structured formats is based on a subtyping relation. When inferring the type of a document, we infer the most specific types of individual values (CSV rows, JSON or XML nodes) and recursively find a common supertype of all child nodes or sample documents.

In this section, we define the *structural type* σ, which is a type of structured data. Note that this type is distinct from the programming language types τ (type providers generate the latter from the former). Next, we define the subtyping relation on structural types σ and describe the algorithm for finding a common supertype.

3.1 Structural types

We distinguish between *non-nullable types* that always have a valid value (written as $\hat{\sigma}$) and *nullable types* that encompass missing and null values (written as σ). We write ν for record field names and $\nu_?$ for names that can be empty:

$$\hat{\sigma} = \nu_? \{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\} \mid \mathsf{float} \mid \mathsf{decimal} \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{string}$$

$$\sigma = \hat{\sigma} \mid \mathsf{option}\langle\hat{\sigma}\rangle \mid [\sigma] \mid \mathsf{any}\langle\sigma_1, \ldots, \sigma_n\rangle \mid \bot \mid \mathsf{null}$$

Non-nullable types include records (consisting of an optional name and fields with their types) and primitive types. Records arising from XML documents are named, while records used by JSON and CSV providers are unnamed.

The three numerical types, int for integers, decimal for small precise decimal numbers and float for floating-point numbers, are related by sub-typing as discussed in §3.2.

Any non-nullable type can be wrapped in the option constructor to explicitly permit the null value. Type providers map option to standard F# option type. A collection type $[\sigma]$ is also nullable and missing values or null are treated as empty collection. The type null is inhabited by the null value (using an overloaded notation) and $\bot$ is the bottom type.

A variant type any is annotated with the types it may represent in the sample document. As discussed earlier (§2.2), variants force the user to handle the case when a value is of a different type than the statically known ones and so it can be seen as a family of top types. For the same reason, the variants also implicitly permit the null value.

3.2 Subtyping relation

Figure 1 provides a basic intuition about the subtyping between structural types. The upper part shows non-nullable types (with records and primitive types) and the lower part shows nullable types with null, collections and optional values. We omit links between the two part, but any type $\hat{\sigma}$ is a subtype of $\mathsf{option}\langle\hat{\sigma}\rangle$ and we abbreviate $\mathsf{option}\langle\sigma\rangle$ as $\sigma?$.

Definition 1. *We write $\sigma_1 :> \sigma_2$ to denote that σ_2 is a subtype of σ_1. The subtyping relation is defined as a transitive reflexive closure of the following rules:*

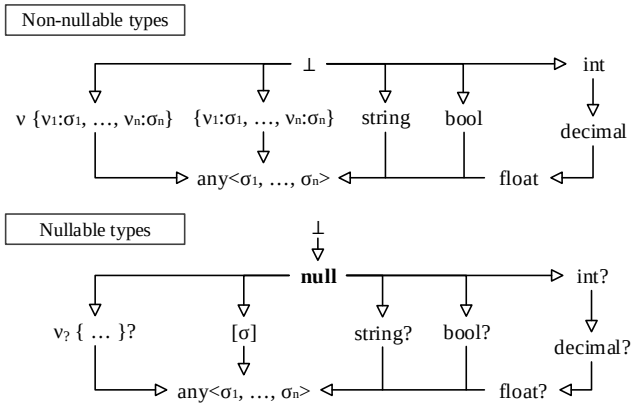

Figure 1. Subtype relation between structural types

Primitives, options, collections

$$\mathsf{float} :> \mathsf{decimal} :> \mathsf{int} \quad \text{(B1)}$$
$$\sigma :> \mathsf{null} \quad (\text{iff } \sigma \neq \hat{\sigma}) \quad \text{(B2)}$$
$$\sigma :> \bot \quad (\text{for all } \sigma) \quad \text{(B3)}$$
$$\mathsf{option}\langle\hat{\sigma}\rangle :> \hat{\sigma} \quad (\text{for all } \hat{\sigma}) \quad \text{(B4)}$$
$$\mathsf{option}\langle\hat{\sigma_1}\rangle :> \mathsf{option}\langle\hat{\sigma_2}\rangle \quad (\text{if } \hat{\sigma_1} :> \hat{\sigma_2}) \quad \text{(B5)}$$
$$[\sigma_1] :> [\sigma_2] \quad (\text{if } \sigma_1 :> \sigma_2) \quad \text{(B6)}$$

Variant types

$$\mathsf{any}\langle\sigma_1, \ldots, \sigma_n\rangle :> \sigma \quad \text{(U1)}$$
$$\mathsf{any}\langle\sigma_1, \ldots, \sigma_n\rangle :> \mathsf{any}\langle\sigma'_1, \ldots, \sigma'_n\rangle \quad \text{(U2)}$$

Record types

$$\nu_? \{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} :> \nu_? \{\nu_1 : \sigma'_1, .., \nu_n : \sigma'_n\} \quad (\text{if } \sigma_i :> \sigma'_i) \quad \text{(R1)}$$

$$\nu_? \{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} :> \nu_? \{\nu_1 : \sigma_1, .., .., \nu_m : \sigma_m\} \quad (\text{when } m \geq n) \quad \text{(R2)}$$

$$\nu_? \{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} :> \nu_? \{\nu_{\pi(1)} : \sigma_{\pi(1)}, .., \nu_{\pi(m)} : \sigma_{\pi(m)}\} \quad (\pi \text{ perm.}) \quad \text{(R3)}$$

$$\nu_? \{\nu_1 : \sigma_1, .., \nu_n : \sigma_n, \nu_{n+1} : \mathsf{null}\} :> \nu_? \{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} \quad \text{(R4)}$$

Here is a summary of the key aspects of the definition:

- Numeric types (B1) ar ordered by the range of values they represent; int is a 32-bit integer, decimal has a range of about -1^{29} to 1^{29} and float is a floating-point number.
- The null type is a subtype of all nullable types (B2), i.e. all types excluding non-nullable types $\hat{\sigma}$. Any non-nullable type is a subtype of its optional version (B4) and both options and collections are covariant (B5, B6).

$$\begin{array}{llll}
\mathsf{tag} = \mathsf{bool} & & \mathsf{tagof}(\mathsf{string}) = \mathsf{string} & \mathsf{tagof}(\mathsf{any}\langle\sigma_1,\ldots,\sigma_n\rangle) = \mathsf{any} \\
\quad \mid \mathsf{string} \mid \mathsf{number} & & \mathsf{tagof}(\mathsf{bool}) = \mathsf{bool} & \mathsf{tagof}(\{\nu_1:\sigma_1,\ \ldots\ ,\nu_n:\sigma_n\}) = \text{rec-anon} \\
\quad \mid \mathsf{any} \mid \mathsf{collection} & & \mathsf{tagof}([\sigma]) = \mathsf{collection} & \mathsf{tagof}(\nu\ \{\nu_1:\sigma_1,\ \ldots\ ,\nu_n:\sigma_n\}) = \text{rec-named } \nu \\
\quad \mid \mathsf{record} \mid \mathsf{named}\ \nu & & \mathsf{tagof}(\hat{\sigma}\ \mathsf{option}) = \mathsf{tagof}(\hat{\sigma}) & \mathsf{tagof}(\sigma) = \mathsf{number} \quad \sigma \in \{\mathsf{int},\mathsf{decimal},\mathsf{float}\}
\end{array}$$

$$\begin{array}{lll}
\lceil\hat{\sigma}\rceil = \hat{\sigma}\ \mathsf{option} & \text{(non-nullable types)} & \lfloor\hat{\sigma}\ \mathsf{option}\rfloor = \hat{\sigma} \quad \text{(option)} \\
\lceil\sigma\rceil = \sigma & \text{(otherwise)} & \lfloor\sigma\rfloor = \sigma \quad \text{(otherwise)}
\end{array}$$

$$\text{(record-1)}\ \frac{(\nu_i = \nu'_j) \Leftrightarrow (i=j) \wedge (i \le k) \qquad \forall i \in \{1..k\}.(\sigma_i \bigtriangledown \sigma'_i \vdash \sigma''_i)}{\begin{array}{l}\nu_? \ \{\nu_1:\sigma_1,\ \ldots\ ,\nu_k:\sigma_k,\ \ldots\ ,\nu_n:\tau_n\} \bigtriangledown \nu_? \ \{\nu'_1:\sigma'_1,\ \ldots\ ,\nu'_k:\sigma'_k,\ \ldots\ ,\nu'_m:\tau'_m\} \vdash \\ \nu_? \ \{\nu_1:\sigma''_1,\ \ldots\ ,\nu_k:\sigma''_k,\nu_{k+1}:\lceil\sigma_{k+1}\rceil,\ldots,\nu_n:\lceil\sigma_n\rceil,\nu'_{k+1}:\lceil\sigma'_{k+1}\rceil,\ldots,\nu'_m:\lceil\sigma'_m\rceil\}\end{array}}$$

$$\text{(var-1)}\ \frac{\exists i.\mathsf{tagof}(\sigma_i) = \mathsf{tagof}(\sigma) \quad \sigma \bigtriangledown \sigma_i \vdash \sigma'_i \quad \mathsf{tagof}(\sigma) \neq \mathsf{any}}{\sigma \bigtriangledown \mathsf{any}\langle\sigma_1,\ldots,\sigma_n\rangle \vdash \mathsf{any}\langle\sigma_1,\ldots,\lfloor\sigma'_i\rfloor,\ldots,\sigma_n\rangle} \qquad \frac{\nexists i.\mathsf{tagof}(\sigma_i) = \mathsf{tagof}(\sigma) \quad \mathsf{tagof}(\sigma) \neq \mathsf{any}}{\sigma \bigtriangledown \mathsf{any}\langle\sigma_1,\ldots,\sigma_n\rangle \vdash \mathsf{any}\langle\sigma_1,\ldots,\sigma_n,\lfloor\sigma\rfloor\rangle}$$

$$\text{(var-2)}\ \frac{(\forall i \in \{1..k\}) \quad \mathsf{tagof}(\sigma_i) = \mathsf{tagof}(\sigma'_i) \quad \sigma_i \bigtriangledown \sigma'_i \vdash \sigma''_i}{\begin{array}{l}\mathsf{any}\langle\sigma_1,\ldots,\sigma_k,\ldots,\sigma_n\rangle \bigtriangledown \mathsf{any}\langle\sigma'_1,\ldots,\sigma'_k,\ldots,\sigma'_m\rangle \vdash \\ \mathsf{any}\langle\sigma''_1,\ldots,\sigma''_k,\sigma_{k+1},\ldots,\sigma_n,\sigma'_{k+1},\ldots,\sigma'_m\rangle\end{array}} \qquad \text{(var-3)}\ \frac{\begin{array}{ll}(\forall i \in \{1,2\}) & \mathsf{tagof}(\sigma_1) \neq \mathsf{tagof}(\sigma_2) \\ \mathsf{tagof}(\sigma_i) \neq \mathsf{any} & \nexists \sigma'_i.(\sigma_i = \mathsf{option}\langle\sigma'_i\rangle)\end{array}}{\sigma_1 \bigtriangledown \sigma_2 \vdash \mathsf{any}\langle\lfloor\sigma_1\rfloor,\lfloor\sigma_2\rfloor\rangle}$$

$$\text{(order-1)}\ \frac{\nu_?\ \{\nu_1:\sigma_1,\ldots,\nu_n:\sigma_n\} \bigtriangledown \sigma \vdash \sigma'}{\nu_?\ \{\nu_{\pi(1)}:\sigma_{\pi(1)},\ldots,\nu_{\pi(n)}:\sigma_{\pi(n)}\} \bigtriangledown \sigma \vdash \sigma'} \qquad \text{(order-2)}\ \frac{\mathsf{any}\langle\sigma_1,\ldots,\sigma_n\rangle \bigtriangledown \sigma \vdash \sigma'}{\mathsf{any}\langle\sigma_{\pi(1)},\ldots,\sigma_{\pi(n)}\rangle \bigtriangledown \sigma \vdash \sigma'} \quad (\pi \text{ perm.})$$

$$\text{(opt)}\ \frac{\hat{\sigma_1} \bigtriangledown \sigma_2 \vdash \sigma}{\mathsf{option}\langle\hat{\sigma_1}\rangle \bigtriangledown \sigma_2 \vdash \mathsf{option}\langle\lfloor\sigma\rfloor\rangle} \qquad \text{(prim)}\ \frac{\sigma_1 :> \sigma_2 \quad \mathsf{tagof}(\sigma_1) = \mathsf{tagof}(\sigma_2) = \mathsf{number}}{\sigma_1 \bigtriangledown \sigma_2 \vdash \sigma_1}$$

$$\text{(list)}\ \frac{\sigma_1 \bigtriangledown \sigma_2 \vdash \sigma}{[\sigma_1] \bigtriangledown [\sigma_2] \vdash [\sigma]} \qquad \text{(sym)}\ \frac{\sigma_1 \bigtriangledown \sigma_2 \vdash \sigma}{\sigma_2 \bigtriangledown \sigma_1 \vdash \sigma}$$

$$\begin{array}{llll}
\text{(refl)} & \sigma \bigtriangledown \sigma \vdash \sigma & \text{(null-1)} & \sigma \bigtriangledown \mathsf{null} \vdash \sigma \quad (\sigma :> \mathsf{null}) \\
\text{(bot)} & \bot \bigtriangledown \sigma \vdash \sigma & \text{(null-2)} & \sigma \bigtriangledown \mathsf{null} \vdash \sigma\ \mathsf{option} \quad (\sigma :\not> \mathsf{null})
\end{array}$$

Figure 2. Inference judgements that define the common supertype relation

- There is a bottom type (B3) and variants behave as top types, because any type σ is a subtype of any variant (U1). There is a range of variants (no *single* top type), but any variant is a subtype of any other variant (U2).
- As usual, the subtyping on records is covariant (R1), subtype can have additional fields (R2) and fields can be reordered (R3). The interesting rule is the last one (R4). Together with covariance, it states that a subtype can omit some fields, provided that their types are nullable.

If we were type-checking programs, defining variants as a family of top types (which are all subtypes of each other) would not add any value over having a single top type. Our purpose is different. We simplify reading structured data and, as shown in §2.2, the types forming a variant provide a valuable information about the sample document.

A particularly important rule is (R4), which allows subtype to have fewer record elements. This allows us to prefer records over variants. For example, given $\{\mathsf{name} : \mathsf{string}\}$ and $\{\mathsf{name}:\mathsf{string},\mathsf{age}:\mathsf{int}\}$, we find a common supertype $\{\mathsf{name}:\mathsf{string},\mathsf{age}:\mathsf{int}\ \mathsf{option}\}$. This is a least upper bound (clarified below) and more usable than another common supertype $\mathsf{var}\langle\{\mathsf{name}:\mathsf{string}\},\{\mathsf{name}:\mathsf{string},\mathsf{age}:\mathsf{int}\}\rangle$.

Some of the aspects of our system (numeric types, handling of null and treatment of missing data) are based on common F# practices and might differ for other languages. For example, OCaml also has multiple numeric types, but would likely always handle null explicitly.

3.3 Common supertype relation

Given two structural types, the *common supertype* relation finds a least upper bound (Theorem 2). When possible, it avoid variant types and prefers records (Corollary 3), which is important for usability as discussed earlier (§2.1).

Definition 2. *A* common supertype *of types σ_1 and σ_2 is a type σ, written $\sigma_1 \bigtriangledown \sigma_2 \vdash \sigma$, obtained according to the inference rules in Figure 2.*

$$\mathsf{getField}(\nu, \nu_i, \nu\,\{\ldots, \nu_i = s_i, \ldots\}) \rightsquigarrow s_i$$
$$\mathsf{getField}(\bullet, \nu_i, \{\ldots, \nu_i = s_i, \ldots\}) \rightsquigarrow s_i$$
$$\mathsf{getField}(\nu, \nu', \nu\,\{\ldots, \nu_i = s_i, \ldots\}) \rightsquigarrow \mathsf{null} \quad (\nexists i.\nu_i = \nu')$$
$$\mathsf{getField}(\bullet, \nu', \{\ldots, \nu_i = s_i, \ldots\}) \rightsquigarrow \mathsf{null} \quad (\nexists i.\nu_i = \nu')$$

$$\mathsf{asDec}(i) \rightsquigarrow d \quad (d = i)$$
$$\mathsf{asDec}(d) \rightsquigarrow d$$

$$\mathsf{asFloat}(i) \rightsquigarrow f \quad (f = i)$$
$$\mathsf{asFloat}(d) \rightsquigarrow f \quad (f = d)$$
$$\mathsf{asFloat}(f) \rightsquigarrow f$$

$$\mathsf{isNull}(\mathsf{null}) \rightsquigarrow \mathsf{true}$$
$$\mathsf{isNull}(\_) \rightsquigarrow \mathsf{false}$$

$$\mathsf{isTag}(\mathsf{string}, t) \rightsquigarrow \mathsf{true}$$
$$\mathsf{isTag}(\mathsf{bool}, s) \rightsquigarrow \mathsf{true} \quad (\text{when } s \in \mathsf{true}, \mathsf{false})$$
$$\mathsf{isTag}(\mathsf{number}, s) \rightsquigarrow \mathsf{true} \quad (\text{when } s = i, s = d \text{ or } s = f)$$
$$\mathsf{isTag}(\mathsf{collection}, [s_1; \ldots; s_n]) \rightsquigarrow \mathsf{true}$$
$$\mathsf{isTag}(\mathsf{collection}, \mathsf{null}) \rightsquigarrow \mathsf{true}$$
$$\mathsf{isTag}(\mathsf{record}, \{\nu_1 \mapsto s_1, \ldots, \nu_n \mapsto s_n\}) \rightsquigarrow \mathsf{true}$$
$$\mathsf{isTag}(\mathsf{named}\ \nu, \nu\,\{\nu_1 \mapsto s_1, \ldots, \nu_n \mapsto s_n\}) \rightsquigarrow \mathsf{true}$$
$$\mathsf{isTag}(\_, \_) \rightsquigarrow \mathsf{false}$$

$$\mathsf{getChildren}([s_1; \ldots; s_n]) \rightsquigarrow [s_1; \ldots; s_n]$$
$$\mathsf{getChildren}(\mathsf{null}) \rightsquigarrow [\,]$$

Figure 3. Reduction rules for conversion functions

When finding a common supertype of records (*record-1*), we return a record that has the union of their fields. The types of shared fields become common supertypes of their respective types while fields that are present in only one record are marked as optional. The (*order-1*) rule lets us reorder fields.

Although all variants are mutual subtypes, we find one that best represents the sample. We avoid nested variants and limit the number of cases. This is done by grouping types that have a common supertype which is not a variant. For example, rather than inferring $\mathsf{any}\langle\mathsf{int}, \mathsf{any}\langle\mathsf{bool}, \mathsf{decimal}\rangle\rangle$, our algorithm finds the common supertype of int and decimal and produces $\mathsf{any}\langle\mathsf{decimal}, \mathsf{bool}\rangle$.

To identify types that have a common supertype which is not a variant, we group the types by a tag. The tag of a type is obtained using a function $\mathsf{tagof}(-) : \sigma \to \mathsf{tag}$. The function is not defined for $\bot$ and null as those are hanled by the (*bot*) and (*null-1*) or (*null-2*) rules, respectively.

The handling of variants is specified using three rules. When combining a non-variant and a variant (*var-1*), the variant may or may not already contain a case with the tag of the other type. If it does, the two types are combined, otherwise a new case is added. When combining two variants (*var-2*), we group the cases that have a shared tags. Finally, (*var-3*) covers the case when we are combining two distinct non-variant types. As unions implicitly permit null values, we use an auxiliary function $\lfloor - \rfloor$ to make nullable types non-nullable (when possible) to simplify the type.

The remaining rules are straightforward. For collections and options, we find the common supertype of the contained value(s); for compatible primitive types, we choose their supertype (*prim*) and a common supertype with null is either the type itself (*null-1*) or an option (*null-2*).

Properties. We say that two types are equivalent if they are mutual subtypes. The partially ordered set of types does not have a *unique* least upper bound, but it does have a least upper bound with respect to this equivalence. The common supertype relation is a function (Theorem 1) and finds the least upper bound (Theorem 2).

Definition 3. *Types σ_1, σ_2 are* equivalent*, written $\sigma_1 \equiv \sigma_2$ iff they are mutual subtypes, i.e. $\sigma_1 <: \sigma_2 \wedge \sigma_1 :> \sigma_2$. An equivalence class of a type σ is $\mathcal{E}(\sigma) = \{\sigma' \mid \sigma \equiv \sigma'\}$.*

The equivalence class $\mathcal{E}$ sheds light on the structure of structural types. It defines a lattice with bottom $\{\bot\}$ and top (set of all variant types). It also joins records with reordered fields (of same names and types) and types that contain other equivalent types (e.g. a records with variants as fields).

Theorem 1 (Function)**.** *For all σ_1 and σ_2 there exists exactly one σ such that $\sigma_1 \triangledown \sigma_2 \vdash \sigma$. Furthermore, if $\sigma_1 \triangledown \sigma_2 \vdash \sigma'$ then it holds that $\sigma :> \sigma'$ and $\sigma' :> \sigma$.*

Proof. The pre-conditions of rules in Figure 2 are disjoint, with the exception of (*order*), (*sym*) and (*refl*). These rules produce types that are subtypes of each other and this property is preserved by rules that use the common supertype relation recursively. □

Theorem 2 (Least upper bound)**.** *If $\sigma_1 \triangledown \sigma_2 \vdash \sigma$ then σ is a least upper bound, i.e. $\sigma :> \sigma_1$ and $\sigma :> \sigma_2$ and for all σ' such that $\sigma' :> \sigma_1$ and $\sigma' :> \sigma_2$, it holds that $\sigma' :> \sigma$.*

Proof. By induction over $\vdash$. Note that the algorithm never produces nested variant or nested option. When one type is variant, the only common supertype is also a variant (*var-2*); for (*var-3*) no other common supertype exists because the two types have distinct tags and are not options. □

Corollary 3. *Given $\sigma_1, \sigma_2, \sigma$ such that $\sigma :> \sigma_1$ and $\sigma :> \sigma_2$ and σ is not a variant, then $\sigma_1 \triangledown \sigma_2 \vdash \sigma'$ and $\sigma' \equiv \sigma$.*

Proof. Consequence of Theorem 2 and the fact that the set of all variants is the top element of $\mathcal{E}$. □

4. Formalising type providers

In this section, we build the necessary theoretical framework for proving relativised type safety of our type providers (§5). We start by discussing the runtime representation of structural values and runtime conversions (§4.1), we embed these into a formal model of an F# subset (§4.2) and we

describe how our type providers turn an inferred structural type into actual F# code (§4.3).

4.1 Structural values and conversions

We represent JSON, XML and CSV documents using the same *structural value*[6]. Structural values are first-order and can be one of the following cases:

$$s = i \mid d \mid f \mid t \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \\ \mid [s_1; \ldots; s_n] \mid \nu_?\{\nu_1 \mapsto s_1, \ldots, \nu_n \mapsto s_n\}$$

The first few cases represent primitive values (i for integers, d for decimals, f for floating point numbers and t for strings) and the missing value null. A collection is written as a list of values in square brackets, separated by semicolons. A record starts with an optional name $\nu_?$, followed by a sequence of field assignments $\nu_i \mapsto s_i$ written in curly brackets.

As indicated by the subtyping, our system permits certain runtime conversions (e.g. an integer 1 can be treated as a floating-point 1.0). The following primitive operations implement the conversions and other helpers:

$$op = \mathsf{asDec}(s) \mid \mathsf{asFloat}(s) \mid \mathsf{getChildren}(s) \\ \mid \mathsf{getField}(\nu?, \nu, s) \mid \mathsf{isNull}(s) \mid \mathsf{isTag}(\mathsf{tag}, s)$$

The type providers generate code that uses these operations at runtime. Their behavior is defined by the reduction rules in Figure 3. The conversion functions (asDec and asFloat) only perform conversions required by the subtyping relation. As noted later (§**??**), our actual implementation is more lax.

The getField operation is used to access a field of a record. The operation ensures that the actual record name matches the expected name (we write • for the name of an unnamed record). The getChildren operation will used to turn null into an empty collction.

Finally, we also define two helper functions. The isNull operation is used to check whether value is null and isTag is used to test whether a value can be treated as a value of a type associated with the specified tag (as defined in Figure 2). The last line defines a "catch all" pattern, which returns false for all remaining cases.

4.2 Featherweight F#

The semantics fragment in the previous section discusses values and operations that are used by F# Data. This section adds a minimal subset of F#. Type providers provide classes and so we focus on the F# object model and combine aspects of standard ML [14] and Featherweight Java [10].

We only need classes with parameter-less members and without inheritance. A class has a single implicit constructor and the declaration closes over constructor parameters. To avoid including all of ML, we only pick constructs for working with options and lists that we need later.

$$\text{(member)}\quad \frac{\mathsf{type}\ C(\overline{x:\tau}) = \ldots\ \mathsf{member}\ N_i : \tau_i = e_i \ldots}{(\mathsf{new}\ C(\overline{v})).N_i \rightsquigarrow e_i[\overline{x} \leftarrow \overline{v}]}$$

$$\text{(cond1)}\quad \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow e_1$$

$$\text{(cond2)}\quad \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow e_2$$

$$\text{(match1)}\quad \mathsf{match\ None\ with\ Some}(x) \to e_1 \mid \mathsf{None} \to e_2 \rightsquigarrow e_2$$

$$\text{(match2)}\quad \mathsf{match\ Some}(v)\ \mathsf{with\ Some}(x) \to e_1 \mid \mathsf{None} \to e_2 \rightsquigarrow e_1[x \leftarrow v]$$

$$\text{(match3)}\quad \mathsf{match}\ [v_1; \ldots; v_n]\ \mathsf{with}\ [x_1; \ldots; x_m] \to e_1 \mid \_ \to e_2 \rightsquigarrow e_2 \quad (m \neq n)$$

$$\text{(match4)}\quad \mathsf{match}\ [v_1; \ldots; v_n]\ \mathsf{with}\ [x_1; \ldots; x_n] \to e_1 \mid \_ \to e_2 \rightsquigarrow e_1[\overline{x} \leftarrow \overline{v}]$$

$$\text{(map)}\quad \mathsf{List.map}\ (\lambda x.e)\ [v_1; \ldots] \rightsquigarrow [e[x \leftarrow v_1]; \ldots]$$

$$\text{(ctx)}\quad E[e] \rightsquigarrow E[e'] \qquad (\text{when } e \rightsquigarrow e')$$

Figure 4. Featherweight F# – Remaining reduction rules

$$\tau = \mathsf{int} \mid \mathsf{decimal} \mid \mathsf{float} \mid \mathsf{bool} \mid \mathsf{string} \\ \mid C \mid \mathsf{StructVal} \mid \mathsf{list}\langle\tau\rangle \mid \mathsf{option}\langle\tau\rangle$$

$$L = \mathsf{type}\ C(\overline{x:\tau}) = \overline{M}$$
$$M = \mathsf{member}\ N : \tau = e$$

$$v = s \mid \mathsf{None} \mid \mathsf{Some}(v) \mid \mathsf{new}\ C(\overline{v}) \mid [v_1; \ldots; v_n]$$
$$e = s \mid op \mid e.N \mid \mathsf{new}\ C(\overline{e}) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\ \mid \mathsf{None} \mid \mathsf{match}\ e\ \mathsf{with\ Some}(x) \to e_1 \mid \mathsf{None} \to e_2 \\ \mid \mathsf{Some}(e) \mid [e_1; \ldots; e_n] \mid \mathsf{List.map}\ (\lambda x \to e_1)\ e_2 \\ \mid \mathsf{match}\ e\ \mathsf{with}\ [x_1; \ldots; x_n] \to e_1 \mid \_ \to e_2$$

The type StructVal is a type of all structural values s. A class definition L consists of a constructor and zero or more members. Values v include previously defined structural values s and values for the option and list type; finally expressions e include previously defined operations op, class construction, member access, conditionals and expressions for working with option values and lists. We include List.map as a special construct to avoid making the language too complex.

Next, we define the reduction relation and (a fragment of) type checking for Featherweight F#. The language presented here is intentionally incomplete. We only define parts needed to prove the relativized safety property (§5).

Reduction. The reduction relation is of the form $e \rightsquigarrow e'$. We also write $e \rightsquigarrow^* e'$ to denote the reflexive and transitive

[6] Here, we diverge from the actual implementation in F# Data which uses a different implementation for each format (reusing existing libraries).

$$\frac{L;\Gamma \vdash e_1 : \mathsf{StructVal}}{L;\Gamma \vdash [e_1;\ldots;e_n] : \mathsf{StructVal}} \qquad \frac{}{L;\Gamma \vdash n : \mathsf{int}}$$

$$\frac{L;\Gamma \vdash e_1 : \tau}{L;\Gamma \vdash [e_1;\ldots;e_n] : \mathsf{list}\langle\tau\rangle} \qquad \frac{}{L;\Gamma \vdash s : \mathsf{StructVal}}$$

$$\frac{L;\Gamma \vdash e : C \quad \mathsf{type}\ C(\overline{x:\tau}) = \ldots\ \mathsf{member}\ N_i : \tau_i = e_i\ \ldots \in L}{L;\Gamma \vdash e.N_i : \tau_i}$$

$$\frac{L;\Gamma \vdash e_i : \tau_i \qquad \mathsf{type}\ C(x_1:\tau_1,\ldots,x_n:\tau_n) = \ldots \in L}{L;\Gamma \vdash \mathsf{new}\ C(e_1,\ldots,e_n) : C}$$

Figure 5. Featherweight F# – Fragment of type checking

closure of $\rightsquigarrow$. The reduction rules for operations *op* were discussed earlier. Figure 4 shows the remaining rules.

The (*ctx*) rule performs a reduction inside a sub-expression specified by an evaluation context. This models the eager evaluation of F#. An evaluation context E is defined as:

$$\begin{aligned} E = {} & [\overline{v}; E; \overline{e}] \mid E.N \mid \mathsf{new}\ C(\overline{v}, E, \overline{e}) \mid \mathsf{Some}(E) \\ & \mid \mathsf{map}\ (\lambda x \to e)\ E \mid \mathsf{if}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid f(E) \\ & \mid \mathsf{match}\ E\ \mathsf{with}\ \mathsf{Some}(x) \to e_1 \mid \mathsf{None} \to e_2 \\ & \mid \mathsf{match}\ E\ \mathsf{with}\ [x_1;\ldots;x_n] \to e_1 \mid \_ \to e_2 \end{aligned}$$

$$f \in \{\mathsf{asDec}, \mathsf{asFloat}, \mathsf{getField}, \mathsf{getChildren}, \mathsf{isNull}, \mathsf{isTag}\}$$

The evaluation first reduces arguments of functions and the evaluation proceeds from left to right as denoted by $\overline{v}, E, \overline{e}$ in constructor arguments or $\overline{v}; E; \overline{e}$ in list initialization.

We write $e[\overline{x} \leftarrow \overline{v}]$ for the result of replacing variables $\overline{x}$ by values $\overline{v}$ in expression. The (*member*) rule reduces a member access using a class definition in the assumption to obtain the body of a member. The remaining six rules give standard reductions for conditionals and pattern matching.

The language is simple, but sufficient for our purpose. All expressions reduce to a value in a finite number of steps or get stuck due to an error condition. An error condition can be a wrong argument passed to conditional, pattern matching or one of the conversion functions from Figure 3.

Type checking. Typing rules in Figure 5 are written using a judgement $L;\Gamma \vdash e : \tau$ where the context also contains a set of class declarations L. The rules demonstrate the key differences from Standard ML and Featherweight Java:

- All structural values s have the type StructVal, but some have other types (Booleans, strings, integers, etc.) as illustrated by the rule for n. For other values, StructVal is the only type – this includes records and null.
- A list containing other structural values $[s_1;\ldots;s_n]$ has a type StructVal, but can also have the $\mathsf{list}\langle\tau\rangle$ type. Conversely, lists that contain non-structural values like objects or options are not of type StructVal.
- Operations *op* (omitted) are treated as functions, accepting StructVal and producing an appropriate result type.
- Rules for checking class construction and member access are similar to corresponding rules of Featherweight Java.

An important part of Featherweight Java that is omitted here is the checking of type declarations (ensuring the bodies of members are well-typed). We omit this, because we consider only classes generated by our type providers (which are correct by construction).

4.3 Type providers

So far, we defined the type inference algorithm which produces a structural type σ from one or more sample documents (§3) and we defined a simplified model of evaluation of F# language (§4.2) and F# Data runtime (§4.3). Next, we define how the type providers work, linking the two parts.

All F# Data type providers take (one or more) sample documents, infer a common supertype σ and then use σ to generate F# types that are exposed to the programmer[7].

Type provider mapping. When generating types, the type provider produces an F# type τ, an expression that wraps the input document (of type StructVal) as a value of type τ and a collection of class definitions. We express it using the following mapping:

$$[\![-]\!]_{\_} : (\sigma \times e) \to (\tau \times e' \times L)$$

The mapping $[\![\sigma]\!]_e$ takes a inferred structural type σ and an expression e, which represents code to obtain a structural value that is being wrapped. It returns an F# type τ, an expression e' which constructs a value of type τ using e and also a set of generated class definitions L.

Figure 6 shows the rules that define $[\![-]\!]_{\_}$. Primitive types are all handled by a single rule. For a given structural type, it returns the corresponding F# type – the types are mapped directly with the exception of bit, which becomes an F# bool. The generated code calls an appropriate conversion function from Figure 3 on the input.

Handling of records is more interesting. We generate a new class C that takes a structural value as constructor parameter. For each record field, we generate a new member with the same name as the field[8] The body of the member calls getField and then passes this expression to $[\![\sigma_i]\!]$ which adds additional wrapping that maps the field (structural value of type σ_i) into an F# type τ_i. The returned expression creates a new instance of C and the mapping returns the class C together with all recursively generated definitions.

A collection type becomes an F# $\mathsf{list}\langle\tau\rangle$. The returned expression calls getChildren (which turns null values into

[7] The actual implementation provides *erased types* as described in [23]. Here, we treat the code as actually generated. This is an acceptable simplification, because F# Data type providers do not rely on laziness that is available through erased types.

[8] The actual F# Data implementation also capitalizes the names.

$$\text{nameof}(\text{string}) = \text{String} \qquad \text{nameof}(\text{number}) = \text{Number} \qquad \text{nameof}(\text{record}) = \text{Record}$$
$$\text{nameof}(\text{bool}) = \text{Boolean} \qquad \text{nameof}(\text{collection}) = \text{List} \qquad \text{nameof}(\text{named } \nu) = \nu$$

$$[\![\sigma_p]\!]_e = \tau_p, op(e), \emptyset \quad \text{where } \tau_p = \sigma_p$$
$$\sigma_p, op \in \{\ (\text{decimal}, \text{asDec}), (\text{float}, \text{asFloat})\ \}$$

$$[\![\ \nu_?\{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\}\ ]\!]_e =$$
$$C, \text{new } C(e), L_1 \cup \ldots \cup L_n \cup \{L\} \quad \text{where}$$
$$C \text{ is a fresh class name}$$
$$L = \text{type } C(v : \text{StructVal}) \ = \ M_1 \ldots M_n$$
$$M_i = \text{member } \nu_i : \tau_i = e_i$$
$$\tau_i, e_i, L_i = [\![\sigma_i]\!]_{e'}, \quad e' = \text{getField}(\nu_?, \nu_i, e)$$

$$[\![\ [\sigma]\ ]\!]_e = \text{list}\langle\tau\rangle, e_b, L \ \text{ where}$$
$$e_b = \text{List.map } (\lambda x \to e')\ (\text{getChildren}(e))$$
$$\tau, e', L = [\![\hat{\sigma}]\!]_x$$

$$[\![\bot]\!]_e = [\![\text{null}]\!]_e = \text{StructVal}, e, \emptyset$$

$$[\![\sigma_p]\!]_e = \tau_p, e, \emptyset \quad \text{where } \tau_p = \sigma_p$$
$$\sigma_p \in \{\ \text{string}, \text{bool}, \text{int}\ \}$$

$$[\![\ \text{any}\langle\sigma_1, \ldots, \sigma_n\rangle\ ]\!]_e =$$
$$C, \text{new } C(e), L_1 \cup \ldots \cup L_n \cup \{L\} \ \text{ where}$$
$$C \text{ is a fresh class name}$$
$$L = \text{type } C(v : \text{StructVal}) \ = \ M_1 \ldots M_n$$
$$M_i = \text{member } \nu_i : \text{option}\langle\tau_i\rangle =$$
$$\text{if isTag}(t_i, v) \text{ then Some}(e_i) \text{ else None}$$
$$\tau_i, e_i, L_i = [\![\sigma_i]\!]_e, \quad t_i = \text{tagof}(\sigma_i), \quad \nu_i = \text{nameof}(t_i)$$

$$[\![\hat{\sigma}\ \text{option}]\!]_e =$$
$$\text{option}\langle\tau\rangle, \text{if isNull } e \text{ then None else Some}(e'), L$$
$$\text{where } \tau, e', L = [\![\hat{\sigma}]\!]_e$$

Figure 6. Type provider – generation of featherweight F# types from inferred structural types

empty lists) and then uses List.map to convert all nested values to an F# type τ. The handling of option type is similar – it checks if the original value is null and if no, it wraps the recursively generated conversion expression e' in the Some constructor.

As discussed earlier, union types are also generated as classes with properties. Given a union type $\sigma_1 + \ldots + \sigma_n$, we get corresponding F# types τ_i and generate n members of type option$\langle\tau_i\rangle$. Each member return Some when the value is not null and has the right structure (checked by isTag). The type inference algorithm also guarantees that there is only one case for each type tag (Section 3.3) and there are no nested union types. Thus, checking for a tag is sufficient and we can also use the tag to identify the name of the generated member (using the nameof function).

Example 1. To illustrate how the type provision mechanism works, we now consider two simple examples. First, assume that the inferred type is a record with two fields (one optional) such as Person { Age : int option, Name : string }. Applying the rules from Figure 6 produces the following class:

```
type Person(v : StructVal) =
  member Age : option⟨int⟩ =
    if isNull (getField(Person, Age, v)) then None
    else Some(asInt(getField(Person, Age, v)))
  member Name : string =
    asStr(getField(Person, Name, v))
```

The body of the Age member is produced by the case for optional types applied to an expression getField(Person, Age, v). If the returned field is not null, then the member calls asInt (produced by the case for primitive types) and wraps the result in the Some constructor. Note that getField is defined even when the field does not exist, but returns null. This lets us treat missing fields as optional fields. The Name member is similar, but does not perform any checks. For completeness, a type corresponding to the record is Person and given a structural value s, we create a Person value by calling new Person(s).

Example 2. The second example illustrates the remaining parts of the type provision, including collections and union types. Reusing the Person type from the previous example, consider a collection [Person + string], which contains a mix of Person and string values.

```
type PersonOrString(v : StructVal) =
  member Person : option⟨Person⟩ =
    if isNull(v) then None else
    if isTag(rec-named Person, v) then
      Some(new Person(v)) else None
  member String : option⟨string⟩ =
    if isNull(v) then None else
    if isTag(string, v) then
      Some(asStr(v)) else None
```

The type provider generates the above type and the collection type is mapped to an F# type list⟨PersonOrString⟩. Given a structural document value s, the code to obtain the wrapped F# value is:

```
List.map (λx → new PersonOrString(x)) (getChildren(s))
```

The PersonOrString type contains one member for each of the union case. In the body, they check that the value is not null and that it has the right structure (using the isTag function). This checks that the value is a record named Person or a string, respectively. If the conditions are satisfied, the value is converted to the F# type corresponding to the case and wrapped in Some.

4.4 Inferring types from values

Before concluding the section on formalizing type providers, there is one more missing piece. The common supertype algorithm in Section 3 discusses the core of the type inference for structural values, but we have not yet clarified how exactly it is used. We address this in the present section.

Given a JSON, XML or CSV document, the F# Data implementation constructs a structural value s from the sample (this is straightforward, but Section 6.1 discusses some interesting aspects). The following defines a mapping $\langle\!\langle - \rangle\!\rangle$ which turns a sample value s into a structural type σ:

$$\begin{array}{rr} \langle\!\langle 0 \rangle\!\rangle = \mathsf{bit} & \langle\!\langle 1 \rangle\!\rangle = \mathsf{bit} \\ \langle\!\langle i \rangle\!\rangle = \mathsf{int} & \langle\!\langle d \rangle\!\rangle = \mathsf{decimal} \\ \langle\!\langle f \rangle\!\rangle = \mathsf{float} & \langle\!\langle t \rangle\!\rangle = \mathsf{string} \\ \langle\!\langle b \rangle\!\rangle = \mathsf{bool} & \langle\!\langle \mathsf{null} \rangle\!\rangle = \mathsf{null} \end{array}$$

$$\langle\!\langle [s_1; \ldots; s_n] \rangle\!\rangle = [\langle\!\langle s_1, \ldots, s_n \rangle\!\rangle]$$

$$\langle\!\langle \nu?\ \{\nu_1 \mapsto s_1, \ldots, \nu_n \mapsto s_n\} \rangle\!\rangle = \nu?\ \{\nu_1 : \langle\!\langle s_1 \rangle\!\rangle, \ldots, \nu_n : \langle\!\langle s_n \rangle\!\rangle\}$$

$$\langle\!\langle s_1, \ldots, s_n \rangle\!\rangle = \sigma_n \quad \text{where}$$
$$\sigma_0 = \top,\ \forall i \in \{1..n\}.\ \sigma_{i-1} \nabla \langle\!\langle s_i \rangle\!\rangle \vdash \sigma_i$$

Primitive values are mapped to their corresponding types, with the exception of 0 and 1, which are inferred as bit. For records, we return a type with field types inferred based on individual values.

The interesting part is inferring type based on multiple samples. We overload the notation and write $\langle\!\langle s_1, \ldots, s_n \rangle\!\rangle$ for a type inferred from multiple samples. This uses the common supertype relation to find a common type for all values (starting with $\top$). This operation is used at the top-level (when calling type provider with multiple samples) and also when inferring the type of a collection.

5. Relativized type safety

Informally, the safety property of F# Data type providers states that, given representative sample documents, any code that can be written using the provided types is guaranteed to work. We call this *relativized safety*, because we cannot avoid *all* errors. In particular, the user can always use the type provider with an input that has a different structure than any of the samples – and in this case, it is expected that the code will fail at runtime (throw an exception in the actual implementation or by getting stuck in our model).

The key question is, what is a representative sample? Given a set of sample documents, the provided type is guaranteed to work if the inferred type of the input is a subtype of any of the sample documents. Going back to Section 3.2, this means that:

- Input can contain smaller numerical values (for example, if a sample contains float, the input can contain an integer).
- Records in the actual input can have additional fields
- Records in the actual input can have fewer fields, provided that the type of the fields is marked as optional in the sample
- Union types in the input can have both fewer or more cases
- When we have a union type in the sample, the actual input can also contain just values of one of the union cases

The following lemma states that the provided code (generated in Figure 6) works correctly on inputs s' that is a subtype of the sample s. More formally, we require that the provided expression (using s' as the input) can be reduced to a value and, if it is a class, all its members can also be reduced to values.

Lemma 4 (Correctness of provided types)**.** *Given a sample value s and an input value s' such that $\langle\!\langle s \rangle\!\rangle :> \langle\!\langle s' \rangle\!\rangle$ and provided type, expression and class declarations $\tau, e, L = [\![\langle\!\langle s \rangle\!\rangle]\!]_{s'}$, then $e \rightsquigarrow^* v$ and if τ is a class ($\tau = C$) then for all members N_i of the class C, it holds that $e.N_i \rightsquigarrow^* v$.*

Proof. By induction over the structure of $[\![-]\!]_e$. For primitives, the conversion functions accept all subtypes. For other cases, analyse the the provided code to see that it can work on all subtypes (*e.g.* getChildren works on null values, getField returns null when a field is missing and, when isTag returns true, then a value has the structure required by the corresponding case). □

Now that we know that the provided types are correct with respect to the subtyping relation, we can look at the main theorem of the paper. It states that, for any input (which is a subtype of any of the samples) and any expression e, a well-typed program that uses the provided types does not "go wrong".

Using the standard approach to syntactic type safety [25], we prove the type preservation (reduction does not change type) and progress (an expression that is not a value can be reduced).

Theorem 5 (Relativized safety)**.** *Assume $s_1, \ldots, s_n$ are samples, $\sigma = \langle\!\langle s_1, \ldots, s_n \rangle\!\rangle$ is an inferred type and $\tau, e, L = [\![\sigma]\!]_x$ are a type, expression and class definitions generated by a type provider.*

Then for all new inputs s' such that $\exists i.(\langle\!\langle s_i \rangle\!\rangle :> \langle\!\langle s' \rangle\!\rangle)$, let $e_s = e[x \leftarrow s']$ be an expression (of type τ) that wraps the input in a provided type. Then, for any expression e_c (user code) such that $\emptyset; y : \tau \vdash e_c : \tau'$ and that does not contain any structural values as sub-expressions, it is the case that $e_c[y \leftarrow e_s] \rightsquigarrow^* v$ for some value v and also $\emptyset; \vdash v : \tau$.

Proof. We discuss the two parts of the proof separately as type preservation (Lemma 6) and progress (Lemma 7). □

Lemma 6 (Preservation)**.** *Given the class definitions L generated by a type provider as specified in the assumptions of Theorem 5, then if $\Gamma \vdash e : \tau$ and $e \rightsquigarrow^* e'$ then $\Gamma \vdash e' : \tau$.*

Proof. By induction over the reduction $\rightsquigarrow$. The cases for the ML subset of Featherweight F# are standard. For (*member*), we check that code generated by type providers in Figure 6 is well-typed. □

The progress lemma states that evaluation of a well-typed program does not reach an undefined state. This is not a problem for the ML subset and object-oriented subset of the calculus. The problematic part are the conversion functions (Figure 3). Given a structural value (which has a type StructVal in our language), the reduction can get stuck if the value does not have a structure required by a specific conversion function used.

The Lemma 4 guarantees that this does not happen. In Theorem 5, we carefully state that we only consider expressions e_c which "[do] not contain any structural values as sub-expressions". This makes sure that the only code working with structural values is the code generated by the type provider.

Lemma 7 (Progress)**.** *Given the assumptions and definitions from Theorem 5, it is the case that $e_c[y \leftarrow e_s] \rightsquigarrow^* v$.*

Proof. By induction over the typing derivation of $L; \emptyset \vdash e_c[y \leftarrow e_s] : \tau'$. The cases for the ML subset are standard. For member access, we rely on Lemma 4. □

5.1 Discussion

The *relativized safety* property does not guarantee the same amount of safety as standard type safety for programming languages without type providers. However, it reflects the reality of programming with external data sources that is increasingly important in the age of web [15]. So, type providers do not reduce the safety – they simply make the existing issues visible.

The actual implementation in F# Data throws an exception for invalid inputs. In contrast, the calculus presented here simply gets stuck. We could extend the calculus with exceptions, but that would obscure the purpose – precisely specifying when the code using type providers does not "go wrong."

As mentioned earlier, *relativized safety* specifies a sufficient, but not a necessary condition. This is also reflected in the conversion functions which allow additional conversions not required by the subtyping relation. In practice, the additional flexibility proves useful (as minor variations in input formats are common). However, an interesting stricter alternative would be to check that an input has the required type when it is loaded – and throw an exception immediately when loading data rather than on member access.

Finally, the formal model presented here ignores two aspects of the real type provider mechanisms. As discussed here, we do not use the *type erasure* mechanism, but treat type providers as if they were actually generating code. For XML, CSV and JSON, both models can be used. However, one interesting aspect of erasing type providers is that the types can be generated lazily – only classes that are actually used in the type checked code are generated. Extending our formalism to capture this aspect is an interesting future work relevant to other type providers – including the World Bank type provider that is also available in the F# Data library.

6. Implementation

The theoretical model in Section 4 presents the core ideas behind the type providers for structured data formats. However, an important part of the success of the F# Data library is also its pragmatic approach, which makes it suitable for use with real-world data.

The formal model discusses some of the pragmatic choices, such as the preference for records over unions and the ability to treat 0 and 1 as both Booleans and numbers. In this section, we briefly discuss some of the remaining practical concerns, starting with the handling of collections.

6.1 Parsing structured data

In this paper, we treat the XML, JSON and CSV formats uniformly as *structural values*. The definition of structural values is, indeed, rich enough to capture all three formats. The structure is similar to JSON (it has primitive values, records and collections). We also added *named* records, which are needed for XML.

- When reading JSON, we directly create a corresponding structural value (with all records unnamed). Optionally, we also read numerical values (and dates) stored in strings.
- When reading CSV, we read each row as an unnamed record and return them in a collection. Optionally, the read values are parsed as numbers or Booleans and missing values become null.
- When reading XML, we create a named record for each node. Attributes become record fields and body becomes a special field named •

The reading of XML documents is perhaps the most interesting. To demonstrate how the body is treated, consider the following:

```
<root id="1">
  <item>Hello!</item>
</root>
```

This XML becomes a record named root with fields id and •. The nested element contains only the • field containing the inner text:

$$\mathsf{root}\ \{\mathsf{id} \mapsto 1, \bullet \mapsto [\mathsf{item}\ \{\bullet \mapsto \text{"Hello!"}\}]\}$$

When generating types for types inferred from XML, we also include a special case to remove the • node using the fact that this can be only a collection (of elements) or a primitive value.

Finally, the XML type provider in F# Data includes an additional option to use *global inference*. In that case, the inference algorithm from Section 4.4 is replaced with an alternative that unifies the types of all records with the same name. This is useful when the sample is, for example, an XHTML document – in this case, all occurrences of an element (*e.g.* `<a>`) are treated as the same type.

6.2 Heterogeneous collections

When introducing the XML type provider (Section 2.2), we briefly mentioned that F# Data implements a special handling of heterogeneous collections. In the example, the provider generated a type with Title member of type string (corresponding to a nested element `<title>` that appears exactly once) and Items member (returning a list of values obtained from multiple `<item>` elements).

To capture this behaviour, we need to extend our earlier treatment of collections. Rather than storing a single type for the elements as in $[\sigma]$, we store multiple possible element types. However, this is not just a union type as we also store *inferred multiplicity* of elements for each of the types:

$$\begin{array}{rl} \psi = & 1 \mid 1? \mid * \\ \sigma = & \ldots \mid [\sigma_1, \psi_1 | \ldots | \sigma_n, \psi_n] \end{array}$$

The multiplicities $1, 1?$ and $*$ represent *exactly one*, *zero or one* and *zero or more*, respectively. Thus, the inferred type of the collection in the RSS feed would be $[\mathsf{title}\ \{\ldots\}\ , 1 |\ \mathsf{item}\ \{\ldots\}, *]$, which reads as a collection containing exactly one title record and any number of item records.

The subtyping relation on heterogeneous collections specifies that a subtype can have fewer cases provided that they do not have to appear exactly once. A subtype can also have a stricter specification of multiplicity (the ordering is $* :> 1? :> 1$).

Finding a common supertype of heterogeneous collections is analogous to the handling of union types. The key rule is (*col*) in Figure 7. It merges cases with the same tag (by merging both the type and the multiplicity). For cases that appear only in one collection, we ensure that the multiplicity is *zero or one* or *zero or more* using the auxiliary definition $\lceil - \rceil$.

6.3 Inferring and generating unions

The F# Data type providers prefer types that do not contain unions (Theorem 3). The motivation for this is twofold. First, the current support for type providers in the F# compiler does not allow type providers to generate F#-specific types (including discriminated unions). This means that the provided type cannot use idiomatic F# representation. Second, when accessing data, using records aids the explorability – when a value is a record, users can type "." and most modern F# editors provide auto-completion list with members.

This choice has an important desirable consequence – it allows treating a union with additional cases as a subtype of another union and, hence, the provided code is guaranteed to work on *more* inputs. Consider a type that is either a Person record or just a string. The type provider exposes it as the following F# class:

```
type PersonOrString =
  member Person : option⟨Person⟩
  member String : option⟨string⟩
```

If we provided an algebraic data type (discriminated union), the consumer would have to use pattern matching that would cover *two* cases. The above type forces the user to also handle a third case, when both properties return None. This also means that union types can silently skip over null values.

That said, exposing a discriminated union (perhaps together with better tooling) is certainly an attractive alternative that we plan to consider in the future. This can also be combined with type inference that chooses between records and union types based on some statistical metric (see related work in Section 7).

6.4 Practical experiences

An important concern about using F# Data in practice is the handling of schema change. When using a type provider, the sample is captured at compile-time. If the schema changes later (so that the actual input is no longer a subtype of the samples), the program using the type provider fails at run-time and it is the developer's responsibility to handle the exception. However, this is the same problem that happens when reading data using any other library.

F# Data can help discover such errors earlier. The example in Section 1 points the JSON type provider to a sample using a live URL. This has the advantage that a recompilation fails when the schema changes, which is an indication that the program needs to be updated to reflect the change. If this is undesirable, it is always possible to cache the sample locally.

In general, there is no better solution for plain XML, CSV and JSON data sources. Some data sources provide versioning support (with meta-data about how the schema changed). For those, a type provider could adapt automatically, but we leave this for future.

An approach that we find useful in practice is to keep a set of samples. When the program fails at run-time (because a service returned data in an unexpected format), the new input can be added as an additional sample, which then shows what parts of code need to be modified.

7. Related and future work

The F# Data library connects two lines of research that have been previously disconnected. The first is extending the

$$\text{(col)}\ \frac{\mathsf{tagof}(\sigma_i) = \mathsf{tagof}(\sigma'_i) \quad \sigma_i \triangledown \sigma'_i \vdash \sigma''_i \quad \phi_i \triangledown \phi'_i \vdash \phi''_i \qquad (\forall i \in \{1..k\})}{\begin{array}{c}[\sigma_1, \psi_1| \ldots |\sigma_k, \psi_k| \ldots |\sigma_n, \psi_n] \triangledown [\sigma'_1, \psi'_1| \ldots |\sigma'_k, \psi'_k| \ldots |\sigma'_m, \psi'_m] \vdash \\ [\sigma''_1, \psi''_1| \ldots |\sigma''_k, \psi''_k|\sigma_{k+1}, \lceil\psi_{k+1}\rceil| \ldots |\sigma_n, \lceil\psi_n\rceil|\sigma'_{k+1}, \lceil\psi'_{k+1}\rceil| \ldots |\sigma'_m, \lceil\psi'_m\rceil]\end{array}}$$

$$\begin{array}{rl} \lceil\phi\rceil = \phi' \text{ such that } \phi \triangledown 1? \vdash \phi' & \\ \phi \triangledown \phi' \vdash \phi & \text{when} \quad \phi :> \phi' \\ \phi \triangledown \phi' \vdash \phi' & \text{when} \quad \phi' :> \phi \end{array}$$

Figure 7. Inference judgement that defines the common supertype of two heterogeneous collections

type systems of programming languages to accommodate external data sources and the second is inferring types for real-world data sources.

The type provider mechanism has been introduced in F# [22, 23], added to Agda [3] and used in areas such as semantic web [17]. Our paper is novel in that it shows the programming language theory behind a concrete type providers.

Extending the type systems. A number of systems integrate external data formats into a programming language. Those include XML [9, 20] and databases [5]. In both of these, the system either requires the user to explicitly define the schema (using the host language) or it has an ad-hoc extension that reads the schema (*e.g.* from a database). LINQ [13] is more general, but relies on code generation when importing the schema.

The work that is most similar to F# Data is the XML and SQL integration in Cω [12]. It extends a C# like object-oriented language with types similar to our structural types (including nullable types, choices with subtyping and heterogeneous collections with multiplicities). However, Cω does not infer the types from samples and extends the type system of the host language (rather than using a general purpose embedding mechanism).

Advanced type systems. Aside from type providers, a number of other advanced type system features could be used to tackle the problem discussed in this paper. The Ur [2] language has a rich system for working with records; meta-programming [18], [6] and multi-stage programming [24] could be used to generate code for the provided types. However, as far as we are aware, none of these systems have been used to provide the same level of integration with XML, CSV and JSON.

Another approach would be to use gradual typing [19, 21] and add types for structured data formats to existing dynamic language to check, for example, JSON manipulation in JavaScript.

Typing real-world data. The second line of research related to our work focuses on inferring structure of real-world data sets. A recent work on JSON [4] infers a succinct type using MapReduce to handle large number of samples. It fuses similar types based on a type similarity measure. This is more sophisticated than our technique, but it would make formally specifying the safety properties (Theorem 5) difficult. Extending our *relativized safety* property to a *probabilistic safety* is an interesting future work.

The PADS project [7, 11] tackles a more general problem of handling *any* data format. The schema definitions in PADS are similar to our structural type. The structure inference for LearnPADS [8] infers the data format from a flat input stream. A PADS type provider could follow many of the patterns we explore in this paper, but formally specifying the safety property would be challenging.

8. Conclusions

In this paper, we explored the F# Data type providers for structured data formats such as XML, CSV and JSON. As most real-world data do not come with an explicit schema, the library uses *type inference* that deduces a type from a set of samples. Next, it uses the *type provider* mechanism to integrate the inferred type directly into the F# type system.

The type inference algorithm we use is based on a common supertype relation. For usability reasons, the algorithm attempts to infer type that does not contain unions and prefers records with optional fields instead. The algorithm is simple and predictable, which is important as developers need to understand how changing the samples affects the resulting types.

In the second part of the paper, we explore the programming language theory behind type providers. F# Data is a prime example of type providers, but our work also demonstrates a more general point. The types generated by type providers can depend on external input (such as samples) and so we can only formulate *relativized safety property*, which says that a program is safe only if the actual inputs satisfy additional conditions – in our case, they have to be subtypes of one of the samples.

The type provider mechanism has been described before, but this paper is novel in that it explores concrete type providers from the perspective of programming language theory. This is particularly important as the F# Data library is becoming de-facto standard tool for data access in F#[9] and other languages are beginning to adopt mechanisms similar to F# type providers.

[9] At the time of writing, the library has 37,000 downloads on NuGet; over 1600 commits and 36 contributors on GitHub.

Acknowledgments

? We would like to thank to the F# Data contributors on GitHub and other colleagues working on type providers, including Jomo Fisher, Keith Battocchi and Kenji Takeda.

References

[1] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. *Acm sigplan notices*, 33(10):183–200, 1998.

[2] A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *ACM Sigplan Notices*, volume 45, pages 122–133. ACM, 2010.

[3] D. R. Christiansen. Dependent type providers. In *Proceedings of Workshop on Generic Programming*, WGP '13, pages 25–34, 2013. ISBN 978-1-4503-2389-5.

[4] D. Colazzo, G. Ghelli, and C. Sartiani. Typing massive json datasets. In *International Workshop on Cross-model Language Design and Implementation*, XLDI '12, 2012.

[5] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer, 2007.

[6] J. Donham and N. Pouillard. Camlp4 and Template Haskell. In *Commercial Users of Functional Programming*, 2010.

[7] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. *ACM Sigplan Notices*, 40(6):295–304, 2005.

[8] K. Fisher, D. Walker, and K. Q. Zhu. LearnPADS: Automatic tool generation from ad hoc data. In *Proceedings of International Conference on Management of Data*, SIGMOD '08, pages 1299–1302, 2008.

[9] H. Hosoya and B. C. Pierce. XDuce: A statically typed xml processing language. *Transactions on Internet Technology*, 3 (2):117–148, 2003.

[10] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. In *ACM SIGPLAN Notices*, volume 34, pages 132–146. ACM, 1999.

[11] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernandez, and A. Gleyzer. PADS/ML: A functional data description language. In *ACM SIGPLAN Notices*, volume 42, pages 77–83. ACM, 2007.

[12] E. Meijer, W. Schulte, and G. Bierman. Unifying tables, objects, and documents. In *Workshop on Declarative Programming in the Context of Object-Oriented Languages*, pages 145–166, 2003.

[13] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET Framework. In *Proceedings of the International Conference on Management of Data*, SIGMOD '06, pages 706–706, 2006.

[14] R. Milner. *The definition of standard ML: revised*. MIT press, 1997.

[15] T. Petricek and D. Syme. In the age of web: Typed functional-first programming revisited. *Submitted to post-proceedings of ML Workshop*, 2014.

[16] T. Petricek, G. Guerra, and Contributors. F# Data: Library for data access, 2015. URL http://fsharp.github.io/FSharp.Data/.

[17] S. Scheglmann, R. Lämmel, M. Leinberger, S. Staab, M. Thimm, and E. Viegas. Ide integrated rdf exploration, access and rdf-based code typing with liteq. In *The Semantic Web: ESWC 2014 Satellite Events*, pages 505–510. Springer, 2014.

[18] T. Sheard and S. P. Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[19] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[20] M. Sulzmann and K. Z. M. Lu. A type-safe embedding of XDuce into ML. *Electr. Notes in Theoretical Comp. Sci.*, 148 (2):239–264, 2006.

[21] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual typing embedded securely in javascript. In *ACM SIGPLAN Notices*, volume 49, pages 425–437. ACM, 2014.

[22] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings of the Workshop on Data Driven Functional Programming*, DDFP'13.

[23] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. Strongly-typed language support for internet-scale information sources. Technical Report MSR-TR-2012-101, Microsoft Research, September 2012.

[24] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Not.*, 32(12):203–217, 1997. ISSN 0362-1340.

[25] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.