Types from data: Making structured data first-class citizens in F#

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 structured formats such as XML, JSON and CSV. Static type systems do not understand such formats, often making data access more cumbersome. Should we give up and leave the messy world of external data to dynamic typing and runtime checks? Of course, not!

We present F# Data, a library that integrates external structured data into F#. As most real-world data does not come with an explicit schema, we develop a shape inference algorithm that infers a shape from representative sample documents. We then integrate the inferred shape into the F# type system using type providers. We formalize the process and prove a relative type soundness theorem.

Our library significantly reduces the amount of data access code and it provides additional safety guarantees when contrasted with the widely used weakly typed techniques.

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

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

1. Introduction

Applications for social networks, finding tomorrow's weather or searching train schedules all communicate with external services. Increasingly, these services provide end-points that return data as CSV, XML or JSON. Most such services do not come with an explicit schema. At best, the documentation provides sample responses for typical requests.

For example, http://openweathermap.org/current contains one example to document an end-point to get the current weather. Using standard libraries, we might call it as[1]:

```
let doc = Http.Request("http://api.owm.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 "Lovely %f!" num
    | _ → failwith "Incorrect format"
  | _ → failwith "Incorrect format"
| _ → failwith "Incorrect format"
```

The code assumes that the response has a particular shape 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 representing the current temperature. When the shape is different, the code fails. While not immediately unsound, the code is prone to errors if strings are misspelled or incorrect shape assumed.

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

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

JsonProvider⟨"..."⟩ invokes a type provider [23] 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.

In short, the *types* come from the sample *data*. In our experience, this technique is both practical and surprisingly effective in achieving more sound information interchange in heterogeneous systems. Our contributions are as follows:

- 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 shape inference algorithm for structured data formats, based on a *preferred shape* relation, that underlies the type providers (§3).
- We give a formal model (§4) and use it to prove *relative type safety* for the type providers (§5).

2. Type providers for structured data

We start with an informal overview that shows how F# Data type providers simplify working with JSON and XML. We introduce the necessary aspects of F# type providers along the way. The examples in this section also illustrate the key design principles of the shape inference algorithm:

[1] We abbreviate the full URL and omit application key (available after registration). The returned JSON is shown in Appendix A and can be used to run the code against a local file.

- The mechanism is predictable (§6.5). The user directly works with the provided types and should understand why a specific type was produced from a given sample.
- The type providers prefer F# object types with properties. This allows extensible (open-world) data formats (§2.2) and it interacts well with developer tooling (§2.1).
- The above makes our techniques applicable to any language with nominal object types (e.g. variations of Java or C# with a type provider mechanism added).
- Finally, we handle practical concerns including support for different numerical types, null and missing data.

The supplementary screencast provides further illustration of the practical developer experience using F# Data.[2]

2.1 Working with JSON documents

The JSON format is a popular data exchange format based on JavaScript data structures. The following is the definition of JsonValue used earlier (§1) to represent JSON data:

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

The earlier example used only a nested record containing a number. To demonstrate other aspects of the JSON type provider, we look at an example that also involves an array:

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

The standard way to print the names and ages would be to pattern match on the parsed JsonValue, check that the top-level node is a Array and iterate over the elements checking that each element is a Record with certain properties. We would throw an exception for values of an incorrect shape. As before, the code would specify field names as strings, which is error prone and can not be statically checked.

Assuming people.json is the above example and data is a string containing JSON of the same shape, we can write:

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

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

We now use a local file as a sample for the type inference, but then processes data from another source. The code achieves a similar simplicity as when using dynamically typed languages, but it is statically type-checked.

Type providers. The notation JsonProvider⟨"people.json"⟩ passes a *static parameter* to the type provider. Static parameters are resolved at compile-time and have to be constant. The provider analyzes the sample and provides a type People. F# editors also execute the type provider at development-time and use the provided types for auto-completion on "." and for background type-checking.

The JsonProvider uses a shape inference algorithm and provides the following F# types for the sample:

```
type Entity =
  member Name : string
  member Age : option⟨float⟩
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. In F#, we use Option.iter to call the specified function (printing) only when an optional value is available. The two age values are an integer 25 and a float 3.5 and so the common inferred type is float. The names of the properties are normalized to follow standard F# naming conventions as discussed later (§6.3).

The type People has two methods for reading data. GetSample parses the sample used for the inference and Parse parses a JSON string. This lets us read data at runtime, provided that it has the same shape as the static sample.

Error handling. In addition to the structure of the types, the type provider also specifies the code of operations such as item.Name. The runtime behaviour is the same as in the earlier hand-written sample (§1) – a member access throws an exception if data does not have the expected shape.

Informally, the safety property (§5) states that if the inputs are compatible with one of the static 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. Moreover, if http://openweathermap.org changes the shape of the response, the code in §1 will not re-compile and the developer knows that the code needs to be corrected.

Objects with properties. The sample code is easy to write thanks to the fact that most F# editors provide auto-completion when "." is typed (see the supplementary screencast). The developer does not need to examine the sample JSON file to see what fields are available. To support this scenario, our type providers map the inferred shapes to F# objects with (possibly optional) properties.

This is demonstrated by the fact that Age becomes an optional member. An alternative is to provide two different record types (one with Name and one with Name and Age),

[2] Available at http://tomasp.net/academic/papers/fsharp-data.

but this would complicate the processing code. It is worth noting that languages with stronger tooling around pattern matching such as Idris [12] might have different preferences.

2.2 Processing XML documents

XML documents are formed by nested elements with attributes. We can view elements as records with a field for each attribute and an additional special field for the nested contents (which is a collection of elements).

Consider a simple extensible document format where a root element `<doc>` can contain a number of document elements, 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 designed primarily to simplify reading of data. For example, say we want to print all headings in the document. The sample shows a part of the document structure (in particular the `<heading>` element), but it does not show all possible elements (say, `<table>`). Assuming the above document is sample.xml, we can write:

```
type Document = XmlProvider⟨"sample.xml"⟩
let root = Document.Load("pldi/another.xml")
for elem in root.Doc do
  Option.iter (printf " - %s") elem.Heading
```

The example iterates over a collection of elements returned by root.Doc. The type of elem provides typed access to elements known statically from the sample and so we can write elem.Heading, which returns an optional string value.

Open world. By its nature, XML is extensible and the sample cannot include all possible nodes.[3] This is the fundamental *open world assumption* about external data. Actual input might be an element about which nothing is known.

For this reason, we do not infer a closed choice between heading, paragraph and image. In the subsequent formalization, we introduce a *top shape* (§3.1) and extend it with labels capturing the statically known possibilities (§3.5). The *labelled top shape* is mapped to the following type:

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

Element is an abstract type with properties. It can represent the statically known elements, but it is not limited to them. For a table element, all three properties would return None.

Using a type with optional properties provides access to the elements known statically from the sample. However the user needs to explicitly handle the case when a value is not a statically known element. In object-oriented languages, the same could be done by providing a class hierarchy, but this loses the easy discoverability when "." is typed.

The provided type is also consistent with our design principles, which prefers optional properties. The gain is that the provided types support both open-world data and developer tooling. It is also worth noting that our shape inference uses labelled top shapes only as the last resort (Lemma 1, §6.4).

2.3 Real-world JSON services

Throughout the introduction, we used data sets that demonstrate the typical problems frequent in the real-world (missing data, inconsistent encoding of primitive values and heterogeneous shapes). The government debt information returned by the World Bank[4] includes all three:

```
[ { "pages": 5 },
  [ { "indicator": "GC.DOD.TOTL.GD.ZS",
      "date": "2012", "value": null },
    { "indicator": "GC.DOD.TOTL.GD.ZS",
      "date": "2010", "value": "35.14229" } ] ]
```

First, the field value is null for some records. Second, numbers in JSON can be represented as numeric literals (without quotes), but here, they are returned as string literals instead.[5] Finally, the top-level element is a collection containing two values of different shape. The record contains meta-data with the total number of pages and the array contains the data. F# Data supports a concept of heterogeneous collection (outlined in in §6.4) and provides the following type:

```
type Record =
  member Pages : int
type Item =
  member Date : int
  member Indicator : string
  member Value : option⟨float⟩
type WorldBank =
  member Record : Record
  member Array : Item[]
```

The inference for heterogeneous collections infers the multiplicities and shapes of nested elements. As there is exactly one record and one array, the provided type WorldBank exposes them as properties Record and Array.

In addition to type providers for JSON and XML, F# Data also implements a type provider for CSV (§6.2). We treat CSV files as lists of records (with field for each column) and so CSV is handled directly by our inference algorithm.

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

[4] Available at http://data.worldbank.org

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

3. Shape inference for structured data

The shape inference algorithm for structured data is based on a shape preference relation. When inferring the shape, it infers the most specific shapes of individual values (CSV rows, JSON or XML nodes) and recursively finds a common shape of all child nodes or all sample documents.

We first define the shape of structured data σ. We use the term *shape* to distinguish shapes of data from programming language *types* τ (type providers generate the latter from the former). Next, we define the preference relation on shapes σ and describe the algorithm for finding a common shape.

The shape algebra and inference presented here is influenced by the design principles we outlined earlier and by the type definitions available in the F# language. The same principles apply to other languages, but details may differ, for example with respect to numerical types and missing data.

3.1 Inferred shapes

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

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

$$\sigma = \hat{\sigma} \mid \mathsf{nullable}\langle\hat{\sigma}\rangle \mid [\sigma] \mid \mathsf{any} \mid \mathsf{null} \mid \bot$$

Non-nullable shapes include records (consisting of a name and fields with their shapes) and primitives. The row variables ρ_i are discussed below. Names of records arising from XML are the names of the XML elements. For JSON records we always use a single name $\bullet$. We assume that record fields can be freely reordered.

We include two numerical primitives, int for integers and float for floating-point numbers. The two are related by the preference relation and we prefer int.

Any non-nullable shape $\hat{\sigma}$ can be wrapped as $\mathsf{nullable}\langle\hat{\sigma}\rangle$ to explicitly permit the null value. Type providers map nullable shapes to the F# option type. A collection $[\sigma]$ is also nullable and null values are treated as empty collections. This is motivated by the fact that a null collection is usually handled as an empty collection by client code. However there is a range of design alternatives (make collections non-nullable or treat null string as an empty string).

The shape null is inhabited by the null value (using an overloaded notation) and $\bot$ is the bottom shape. The any shape is the top shape, but we later add labels for statically known alternative shapes (§3.5) as discussed earlier (§2.2).

During inference we use row-variables ρ_i [1] in record shapes to represent the flexibility arising from records in samples. For example, when a record $\mathsf{Point}\ \{\mathsf{x} \mapsto 3\}$ occurs in a sample, it may be combined with $\mathsf{Point}\ \{\mathsf{x} \mapsto 3, \mathsf{y} \mapsto 4\}$ that contains more fields. The overall shape inferred must account for the fact that any extra fields are optional, giving an inferred shape $\mathsf{Point}\ \{\mathsf{x} : \mathsf{int},\ \mathsf{y} : \mathsf{nullable}\langle\mathsf{int}\rangle\}$.

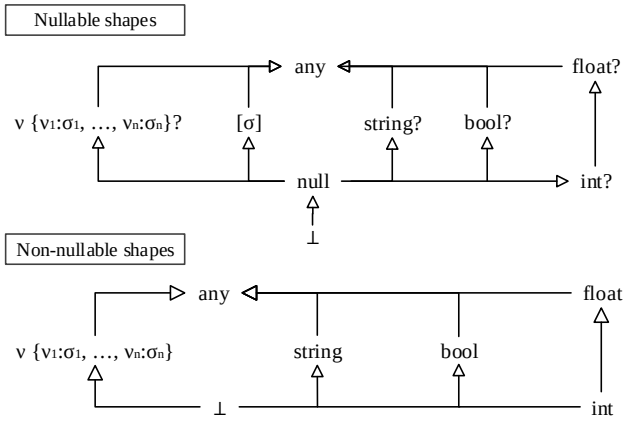

Figure 1. Important aspects of the preferred shape relation

3.2 Preferred shape relation

Figure 1 provides an intuition about the preference between shapes. The lower part shows non-nullable shapes (with records and primitives) and the upper part shows nullable shapes with null, collections and nullable shapes. In the diagram, we abbreviate $\mathsf{nullable}\langle\sigma\rangle$ as $\sigma?$ and we omit links between the two parts; a shape $\hat{\sigma}$ is preferred over $\mathsf{nullable}\langle\hat{\sigma}\rangle$.

Definition 1. *For ground σ_1, σ_2 (i.e. without ρ_i variables), we write $\sigma_1 \sqsubseteq \sigma_2$ to denote that σ_1 is preferred over σ_2. The shape preference relation is defined as a transitive reflexive closure of the following rules:*

$$\mathsf{int} \sqsubseteq \mathsf{float} \tag{1}$$

$$\mathsf{null} \sqsubseteq \sigma \quad (\text{for } \sigma \neq \hat{\sigma}) \tag{2}$$

$$\hat{\sigma} \sqsubseteq \mathsf{nullable}\langle\hat{\sigma}\rangle \quad (\text{for all } \hat{\sigma}) \tag{3}$$

$$\mathsf{nullable}\langle\hat{\sigma_1}\rangle \sqsubseteq \mathsf{nullable}\langle\hat{\sigma_2}\rangle \quad (\text{if } \hat{\sigma_1} \sqsubseteq \hat{\sigma_2}) \tag{4}$$

$$[\sigma_1] \sqsubseteq [\sigma_2] \quad (\text{if } \sigma_1 \sqsubseteq \sigma_2) \tag{5}$$

$$\bot \sqsubseteq \sigma \quad (\text{for all } \sigma) \tag{6}$$

$$\sigma \sqsubseteq \mathsf{any} \tag{7}$$

$$\nu\ \{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} \sqsubseteq \nu\ \{\nu_1 : \sigma'_1, .., \nu_n : \sigma'_n\} \quad (\text{if } \sigma_i \sqsubseteq \sigma'_i) \tag{8}$$

$$\nu\ \{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} \sqsubseteq \nu\ \{\nu_1 : \sigma_1, .., .., \nu_m : \sigma_m\} \quad (\text{when } m \leq n) \tag{9}$$

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

- Numeric shape with smaller range is preferred (1) and we choose 32-bit int over float when possible.
- The null shape is preferred over all nullable shapes (2), i.e. all shapes excluding non-nullable shapes $\hat{\sigma}$. Any non-nullable shape is preferred over its nullable version (3)
- Nullable shapes and collections are covariant (4, 5).
- There is a bottom shape (6) and any behaves as the top shape, because any shape σ is preferred over any (7).
- The record shapes are covariant (8) and preferred record can have additional fields (9).

$$
\begin{array}{rcll}
\mathsf{csh}(\sigma, \sigma) & = & \sigma & (\textit{eq}) \\
\mathsf{csh}([\sigma_1], [\sigma_2]) & = & [\mathsf{csh}(\sigma_1, \sigma_2)] & (\textit{list}) \\
\mathsf{csh}(\bot, \sigma) = \mathsf{csh}(\sigma, \bot) & = & \sigma & (\textit{bot}) \\
\mathsf{csh}(\mathsf{null}, \sigma) = \mathsf{csh}(\sigma, \mathsf{null}) & = & \lceil \sigma \rceil & (\textit{null}) \\
\mathsf{csh}(\mathsf{any}, \sigma) = \mathsf{csh}(\sigma, \mathsf{any}) & = & \mathsf{any} & (\textit{top}) \\
\mathsf{csh}(\mathsf{float}, \mathsf{int}) = \mathsf{csh}(\mathsf{int}, \mathsf{float}) & = & \mathsf{float} & (\textit{num}) \\
\mathsf{csh}(\sigma_2, \mathsf{nullable}\langle\hat{\sigma_1}\rangle) = \mathsf{csh}(\mathsf{nullable}\langle\hat{\sigma_1}\rangle, \sigma_2) & = & \lceil \mathsf{csh}(\hat{\sigma_1}, \sigma_2) \rceil & (\textit{opt}) \\
\mathsf{csh}(\nu\ \{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\}, \nu\ \{\nu_1 : \sigma'_1, \ldots, \nu_n : \sigma'_n\}) & = & \nu\ \{\nu_1 : \mathsf{csh}(\sigma_1, \sigma'_1), \ldots, \nu_n : \mathsf{csh}(\sigma_n, \sigma'_n)\} & (\textit{recd}) \\
\mathsf{csh}(\sigma_1, \sigma_2) & = & \mathsf{any} \quad (\text{when } \sigma_1 \neq \nu\ \{\ldots\} \text{ or } \sigma_2 \neq \nu\ \{\ldots\}) & (\textit{any})
\end{array}
$$

$$
\begin{array}{lll}
\lceil \hat{\sigma} \rceil = \mathsf{nullable}\langle\hat{\sigma}\rangle & \text{(non-nullable shapes)} & \qquad \lfloor \mathsf{nullable}\langle\hat{\sigma}\rangle \rfloor = \hat{\sigma} \quad \text{(nullable shape)} \\
\lceil \sigma \rceil = \sigma & \text{(otherwise)} & \qquad \lfloor \sigma \rfloor = \sigma \quad \text{(otherwise)}
\end{array}
$$

Figure 2. The rules that define the common preferred shape function

3.3 Common preferred shape relation

Given two ground shapes, the *common preferred shape* is the least upper bound of the shape with respect to the preferred shape relation. The least upper bound prefers records, which is important for usability as discussed earlier (§2.2).

Definition 2. *A* common preferred shape *of two ground shapes σ_1 and σ_2 is a shape* $\mathsf{csh}(\sigma_1, \sigma_2)$ *obtained according to Figure 2. The rules are matched from top to bottom.*

The fact that the rules of csh are matched from top to bottom resolves the ambiguity between certain rules. Most importantly (*any*) is used only as the last resort.

When finding a common shape of two records (*recd*) we find common preferred shapes of their respective fields. We can find a common shape of two different numbers (*num*); for two collections, we combine their elements (*list*). When one shape is nullable (*opt*), we find the common non-nullable shape and ensure the result is nullable using $\lceil - \rceil$, which is also applied when one of the shapes is null (*null*).

When defined, csh finds the unique least upper bound of the partially ordered set of ground shapes (Lemma 1).

Lemma 1 (Least upper bound)**.** *For ground σ_1 and σ_2, if* $\mathsf{csh}(\sigma_1, \sigma_2) \vdash \sigma$ *then σ is a least upper bound by $\sqsupseteq$.*

Proof. By induction over the structure of the shapes σ_1, σ_2. Note that csh only infers the top shape any when on of the shapes is the top shape (*top*) or when there is no other option (*any*); a nullable shape is introduced in $\lceil - \rceil$ only when no non-nullable shape can be used (*null*), (*opt*). □

3.4 Inferring shapes from samples

We now specify how we obtain the shape from data. As clarified later (§6.2), we represent JSON, XML and CSV documents using the same first-order *data* value:

$$
\begin{array}{l}
d = i \mid f \mid s \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \\
\quad \mid [d_1; \ldots; d_n] \mid \nu\ \{\nu_1 \mapsto d_1, \ldots, \nu_n \mapsto d_n\}
\end{array}
$$

The definition includes primitive values (i for integers, f for floats and s for strings) and null. A collection is written as a list of values in square brackets. A record starts with a name ν, followed by a sequence of field assignments $\nu_i \mapsto d_i$.

Figure 3 defines a mapping $S(d_1, \ldots, d_n)$ which turns a collection of sample data $d_1, \ldots, d_n$ into a shape σ. Before applying S, we assume each record in each d_i is marked with a fresh row inference variable ρ_i. We then choose a ground, minimal substitution θ for row variables. Because ρ_i variables represent potentially missing fields, the $\lceil - \rceil$ operator from Figure 2 is applied to all types in the vector.

This is sufficient to equate the record field labels and satisfy the pre-conditions in rule (*recd*) when multiple record shapes are combined. The csh function is not defined for two records with mis-matching fields, however, the fields can always be made to match, through a substitution for row variables. In practice, θ is found via row variable unification [17]. We omit the details here. No ρ_i variables remain after inference as the substitution chosen is ground.

Primitive values are mapped to their corresponding shapes. When inferring a shape from multiple samples, we use the common preferred shape relation to find a common shape for all values (starting with $\bot$). This operation is used when calling a type provider with multiple samples and also when inferring the shape of collection values.

$$
\begin{array}{lll}
S(i) = \mathsf{int} & S(\mathsf{null}) = \mathsf{null} & S(\mathsf{true}) = \mathsf{bool} \\
S(f) = \mathsf{float} & S(s) = \mathsf{string} & S(\mathsf{false}) = \mathsf{bool}
\end{array}
$$

$$
S([d_1; \ldots; d_n]) = [S(d_1, \ldots, d_n)]
$$

$$
\begin{array}{l}
S(\nu\ \{\nu_1 \mapsto d_1, \ldots, \nu_n \mapsto d_n\}_{\rho_i}) = \\
\quad \nu\ \{\nu_1 : S(d_1), \ldots, \nu_n : S(d_n), \lceil \theta(\rho_i) \rceil\}
\end{array}
$$

$$
\begin{array}{l}
S(d_1, \ldots, d_n) = \sigma_n \quad \text{where} \\
\quad \sigma_0 = \bot,\ \forall i \in \{1..n\}.\ \sigma_{i-1} \triangledown S(d_i) \vdash \sigma_i
\end{array}
$$

Choose minimal θ by ordering $\sqsubseteq$ lifted over substitutions

Figure 3. Shape inference from sample data

$$\begin{array}{rcl} \mathsf{tag} & = & \mathsf{collection} \mid \mathsf{number} \\ & \mid & \mathsf{nullable} \mid \mathsf{string} \\ & \mid & \nu \mid \mathsf{any} \mid \mathsf{bool} \end{array}$$

$$\begin{array}{rl} \mathsf{tagof}(\mathsf{string}) = \mathsf{string} & \mathsf{tagof}(\mathsf{any}\langle\sigma_1,\ldots,\sigma_n\rangle) = \mathsf{any} \\ \mathsf{tagof}(\mathsf{bool}) = \mathsf{bool} & \mathsf{tagof}(\nu\ \{\nu_1:\sigma_1,\ \ldots\ ,\nu_n:\sigma_n\}) = \nu \\ \mathsf{tagof}(\mathsf{int}) = \mathsf{number} & \mathsf{tagof}(\mathsf{nullable}\langle\hat{\sigma}\rangle) = \mathsf{nullable} \\ \mathsf{tagof}(\mathsf{float}) = \mathsf{number} & \mathsf{tagof}([\sigma]) = \mathsf{collection} \end{array}$$

$$\mathsf{csh}(\mathsf{any}\langle\sigma_1,\ldots,\sigma_k,\ldots,\sigma_n\rangle, \mathsf{any}\langle\sigma'_1,\ldots,\sigma'_k,\ldots,\sigma'_m\rangle) = \mathsf{any}\langle\mathsf{csh}(\sigma_1,\sigma'_1),\ldots,\mathsf{csh}(\sigma_k,\sigma'_k),\sigma_{k+1},\ldots,\sigma_n,\sigma'_{k+1},\ldots,\sigma'_m\rangle \quad \textit{(top-merge)}$$
For i,j such that $(\mathsf{tagof}(\sigma_i) = \mathsf{tagof}(\sigma'_j)) \Leftrightarrow (i=j) \wedge (i \leq k)$

$$\mathsf{csh}(\sigma, \mathsf{any}\langle\sigma_1,\ldots,\sigma_n\rangle) = \mathsf{csh}(\mathsf{any}\langle\sigma_1,\ldots,\sigma_n\rangle, \sigma) = \mathsf{any}\langle\sigma_1,\ldots,\lfloor\mathsf{csh}(\sigma,\sigma_i)\rfloor,\ldots,\sigma_n\rangle \quad \textit{(top-incl)}$$
For i such that $\mathsf{tagof}(\sigma_i) = \mathsf{tagof}(\lfloor\sigma\rfloor)$

$$\mathsf{csh}(\sigma, \mathsf{any}\langle\sigma_1,\ldots,\sigma_n\rangle) = \mathsf{any}\langle\sigma_1,\ldots,\sigma_n,\lfloor\sigma\rfloor\rangle \quad \textit{(top-add)}$$

$$\mathsf{csh}(\sigma_1,\sigma_2) = \mathsf{any}\langle\lfloor\sigma_1\rfloor,\lfloor\sigma_2\rfloor\rangle \quad \textit{(top-any)}$$

Figure 4. Extending the common preferred shape relation for labelled top shapes

3.5 Adding labelled top shapes

When analyzing the structure of shapes, it suffices to consider a single top shape any. The type providers need more information to provide typed access to the possible alternative shapes of data, such as XML nodes.

We extend the core model (sufficient for the discussion of relative safety) with *labelled top shapes* defined as:

$$\sigma = \ldots \mid \mathsf{any}\langle\sigma_1,\ldots,\sigma_n\rangle$$

The shapes $\sigma_1,\ldots,\sigma_n$ represent statically known shapes that appear in the sample and that we expose in the provided type. As discussed earlier (§2.2) this is important when reading external *open world* data. The labels do not affect the preferred shape relation and $\mathsf{any}\langle\sigma_1,\ldots,\sigma_n\rangle$ should still be seen as the top shape, regardless of the labels[6].

The common preferred shape function is extended to find a labelled top shape that best represents the sample. The new rules for any appear in Figure 4. We define shape *tags* to identify shapes that have a common preferred shape which is not the top shape. We use it to limit the number of labels and avoid nesting by grouping shapes by the shape tag. Rather than inferring $\mathsf{any}\langle\mathsf{int}, \mathsf{any}\langle\mathsf{bool},\mathsf{float}\rangle\rangle$, our algorithm joins int and float and produces $\mathsf{any}\langle\mathsf{float},\mathsf{bool}\rangle$.

When combining two top shapes (*top-merge*), we group the annotations by their tags. When combining a top with another shape, the labels may or may not already contain a case with the tag of the other shape. If they do, the two shapes are combined (*top-incl*), otherwise a new case is added (*top-add*). Finally, (*top-all*) replaces earlier (*any*) and combines two distinct non-top shapes. As top shapes implicitly permit null values, we make the labels non-nullable using $\lfloor-\rfloor$.

The revised algorithm still finds a shape which is the least upper bound. This means that labelled top shape is only inferred when there is no other alternative.

Stating properties of the labels requires refinements to the *preferred shape* relation. We leave the details to future work, but we note that the algorithm infers the best labels in the sense that there are labels that enable typed access to every possible value in the sample, but not more. The same is the case for nullable fields of records.

4. Formalizing type providers

This section presents the formal model of F# Data integration. To represent the programming language that hosts the type provider, we introduce the Foo calculus, a subset of F# with objects and properties, extended with operations for working with weakly typed structured data along the lines of the F# Data runtime. Finally, we describe how type providers turn inferred shapes into Foo classes (§4.2).

$$\begin{array}{rcl} \tau & = & \mathsf{int} \mid \mathsf{float} \mid \mathsf{bool} \mid \mathsf{string} \mid C \mid \mathsf{Data} \\ & \mid & \tau_1 \to \tau_2 \mid \mathsf{list}\langle\tau\rangle \mid \mathsf{option}\langle\tau\rangle \\ L & = & \mathsf{type}\ C(\overline{x:\tau}) = \overline{M} \\ M & = & \mathsf{member}\ N : \tau = e \\ v & = & d \mid \mathsf{None} \mid \mathsf{Some}(v) \mid \mathsf{new}\ C(\overline{v}) \mid v_1 :: v_2 \\ e & = & d \mid op \mid e_1\ e_2 \mid \lambda x.e \mid e.N \mid \mathsf{new}\ C(\overline{e}) \\ & \mid & \mathsf{None} \mid \mathsf{match}\ e\ \mathsf{with}\ \mathsf{Some}(x) \to e_1 \mid \mathsf{None} \to e_2 \\ & \mid & \mathsf{Some}(e) \mid e_1 = e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{nil} \\ & \mid & e_1 :: e_2 \mid \mathsf{match}\ e\ \mathsf{with}\ x_1 :: x_2 \to e_1 \mid \mathsf{nil} \to e_2 \\ op & = & \mathsf{convFloat}(\sigma, e) \mid \mathsf{convPrim}(\sigma, e) \\ & \mid & \mathsf{convField}(\nu_1,\nu_2,e,e) \mid \mathsf{convNull}(e_1,e_2) \\ & \mid & \mathsf{convElements}(e_1,e_2) \mid \mathsf{hasShape}(\sigma,e) \end{array}$$

Figure 5. The syntax of the Foo calculus

[6] An alternative would be to add unions of shapes, but doing so in a way that is compatible with the open-world assumption breaks the existence of unique lower bound of the preferred shape relation.

Part I. Reduction rules for conversion functions

$$\mathsf{hasShape}(\nu\ \{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\}, \nu'\ \{\nu'_1 \mapsto d_1, \ldots, \nu'_m \mapsto d_m\}) \rightsquigarrow (\nu = \nu') \wedge$$
$$(\ ((\nu_1 = \nu'_1) \wedge \mathsf{hasShape}(\sigma_1, d_1)) \vee \ldots \vee ((\nu_1 = \nu'_m) \wedge \mathsf{hasShape}(\sigma_1, d_m)) \vee \ldots \vee$$
$$((\nu_n = \nu'_1) \wedge \mathsf{hasShape}(\sigma_n, d_1)) \vee \ldots \vee ((\nu_n = \nu'_m) \wedge \mathsf{hasShape}(\sigma_n, d_m))\ )$$

$$\mathsf{hasShape}([\sigma], [d_1; \ldots; d_n]) \rightsquigarrow \mathsf{hasShape}(\sigma, d_1) \wedge \ldots \wedge \mathsf{hasShape}(\sigma, d_n)$$
$$\mathsf{hasShape}([\sigma], \mathsf{null}) \rightsquigarrow \mathsf{true}$$

$$\mathsf{hasShape}(\mathsf{string}, s) \rightsquigarrow \mathsf{true}$$
$$\mathsf{hasShape}(\mathsf{int}, i) \rightsquigarrow \mathsf{true}$$
$$\mathsf{hasShape}(\mathsf{bool}, d) \rightsquigarrow \mathsf{true} \quad (\text{when } d \in \mathsf{true}, \mathsf{false})$$
$$\mathsf{hasShape}(\mathsf{float}, d) \rightsquigarrow \mathsf{true} \quad (\text{when } d = i \text{ or } d = f)$$
$$\mathsf{hasShape}(\_, \_) \rightsquigarrow \mathsf{false}$$

$$\mathsf{convFloat}(\mathsf{float}, i) \rightsquigarrow f\ (f = i)$$
$$\mathsf{convFloat}(\mathsf{float}, f) \rightsquigarrow f$$

$$\mathsf{convNull}(\mathsf{null}, e) \rightsquigarrow \mathsf{None}$$
$$\mathsf{convNull}(d, e) \rightsquigarrow \mathsf{Some}(e\ d)$$

$$\mathsf{convPrim}(\sigma, d) \rightsquigarrow d \quad (\sigma, d \in \{(\mathsf{int}, i), (\mathsf{string}, s), (\mathsf{bool}, b)\})$$
$$\mathsf{convField}(\nu, \nu_i, \nu\ \{\ldots, \nu_i = d_i, \ldots\}, e) \rightsquigarrow e\ d_i$$
$$\mathsf{convField}(\nu, \nu', \nu\ \{\ldots, \nu_i = d_i, \ldots\}, e) \rightsquigarrow e\ \mathsf{null} \quad (\nexists i. \nu_i = \nu')$$
$$\mathsf{convElements}([d_1; \ldots; d_n], e) \rightsquigarrow e\ d_1 :: \ldots :: e\ d_n :: \mathsf{nil}$$
$$\mathsf{convElements}(\mathsf{null}) \rightsquigarrow \mathsf{nil}$$

Part II. Reduction rules for the rest of the Foo calculus

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

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

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

$$\text{(eq1)}\ v = v' \rightsquigarrow \mathsf{true} \quad (\text{when } v = v')$$

$$\text{(eq2)}\ v = v' \rightsquigarrow \mathsf{false} \quad (\text{when } v \neq v')$$

$$\text{(fun)}\ (\lambda x.e)\ v \rightsquigarrow e[x \leftarrow v]$$

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

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

$$\text{(match3)}\ \mathsf{match}\ \mathsf{nil}\ \mathsf{with}\ x_1 :: x_2 \rightarrow e_1 \mid \mathsf{nil} \rightarrow e_2 \rightsquigarrow e_2$$

$$\text{(match4)}\ \mathsf{match}\ v_1 :: v_2\ \mathsf{with}\ x_1 :: x_2 \rightarrow e_1 \mid \mathsf{nil} \rightarrow e_2 \rightsquigarrow e_1[\overline{x} \leftarrow \overline{v}]$$

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

Figure 6. Foo – Reduction rules for the Foo calculus and dynamic data operations

Type providers for structured data map the "dirty" world of weakly typed structured data into a "nice" world of strong types. To model this, the Foo calculus does not have null values and data values d are never directly exposed. Furthermore Foo is simply typed: despite using class types and object notation for notational convenience, it has no subtyping.

4.1 The Foo calculus

The syntax of the calculus is shown in Figure 5. The type Data is the type of structural data d. A class definition L consists of a single constructor and zero or more parameterless members. The declaration implicitly closes over the constructor parameters. Values v include previously defined data d; expressions e include class construction, member access, usual functional constructs (functions, lists, options) and conditionals. The *op* constructs are discussed next.

Dynamic data operations. The Foo programs can only work with Data values using certain primitive operations. Those are modelled by the *op* primitives. In F# Data, those are internal and users never access them directly.

The behaviour of the dynamic data operations is defined by the reduction rules in Figure 6 (Part I). The typing is shown in Figure 7 and is discussed later. The hasShape function represents a runtime shape test. It checks whether a Data value d (Section 3.4) passed as the second argument has a shape specified by the first argument. For records, we have to check that for each field $\nu_1, \ldots, \nu_n$ in the record, the actual record value has a field of the same name with a matching shape. The last line defines a "catch all" pattern, which returns false for all remaining cases. We treat $e_1 \vee e_2$ and $e_1 \wedge e_2$ as a syntactic sugar for if..then..else so the result of the reduction is just a Foo expression.

The remaining operations convert data values into values of less preferred shape. The convPrim and convFloat operations take the required shape and a data value. When the data does not match the required type, they do not reduce. For example, convPrim(bool, 42) represents a stuck state, but convFloat(float, 42) turns an integer 42 into a floating-point numerical value 42.0.

The convNull, convElements and convField operations take an additional parameter e which represents a function to be used in order to convert a contained value (non-null optional value, list elements or field value); convNull turns null data value into None and convElements turns a data collection $[d_1, \ldots, d_n]$ into a Foo list $v_1 :: \ldots :: v_n :: \mathsf{nil}$ and a null value into an empty list.

$$\frac{}{L;\Gamma \vdash d : \mathsf{Data}} \quad \frac{}{L;\Gamma \vdash i : \mathsf{int}} \quad \frac{}{L;\Gamma \vdash f : \mathsf{float}} \quad \frac{L;\Gamma, x : \tau_1 \vdash e : \tau_2}{L;\Gamma \vdash \lambda x.e : \tau_2} \quad \frac{L;\Gamma \vdash e_2 : \tau_1 \quad L;\Gamma \vdash e_1 : \tau_1 \to \tau_2}{L;\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{L;\Gamma \vdash e : \mathsf{Data}}{L;\Gamma \vdash \mathsf{hasShape}(\sigma, e) : \mathsf{bool}} \quad \frac{L;\Gamma \vdash e : \mathsf{Data} \quad \tau \in \{\mathsf{int}, \mathsf{float}\}}{L;\Gamma \vdash \mathsf{convFloat}(\sigma, e) : \mathsf{float}} \quad \frac{L;\Gamma \vdash e_1 : \mathsf{Data} \quad L;\Gamma \vdash e_2 : \mathsf{Data} \to \tau}{L;\Gamma \vdash \mathsf{convNull}(e_1, e_2) : \mathsf{option}\langle\tau\rangle}$$

$$\frac{L;\Gamma \vdash e : \mathsf{Data} \quad \mathsf{prim} \in \{\mathsf{int}, \mathsf{string}, \mathsf{bool}\}}{L;\Gamma \vdash \mathsf{convPrim}(\mathsf{prim}, e) : \mathsf{prim}} \quad \frac{L;\Gamma \vdash e_1 : \mathsf{Data} \quad L;\Gamma \vdash e_2 : \mathsf{Data} \to \tau}{L;\Gamma \vdash \mathsf{convElements}(e_1, e_2) : \mathsf{list}\langle\tau\rangle} \quad \frac{L;\Gamma \vdash e_1 : \mathsf{Data} \quad L;\Gamma \vdash e_2 : \mathsf{Data} \to \tau}{L;\Gamma \vdash \mathsf{convField}(\nu, \nu', e_1, e_2) : \tau}$$

$$\frac{L;\Gamma \vdash e : C \quad \mathsf{type}\ C(\overline{x : \tau}) = \ldots\ \mathsf{member}\ N_i : \tau_i = e_i\ \ldots \in L}{L;\Gamma \vdash e.N_i : \tau_i} \quad \frac{L;\Gamma \vdash e_i : \tau_i \quad \mathsf{type}\ C(x_1 : \tau_1, \ldots, x_n : \tau_n) = \ldots \in L}{L;\Gamma \vdash \mathsf{new}\ C(e_1, \ldots, e_n) : C}$$

Figure 7. Foo – Fragment of type checking

Reduction. The reduction relation is of the form $L, e \rightsquigarrow e'$. We omit class declarations L where implied by the context and write $e \rightsquigarrow^* e'$ for the reflexive, transitive closure of $\rightsquigarrow$.

Figure 6 (Part II) shows the reduction rules. The (*member*) rule reduces a member access using a class definition in the assumption. The (*ctx*) rule models the eager evaluation of F# and performs a reduction inside a sub-expression specified by an evaluation context E:

$$\begin{aligned} E = {} & v :: E \mid v\ E \mid E.N \mid \mathsf{new}\ C(\overline{v}, E, \overline{e}) \\ & \mid \mathsf{if}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid E = e \mid v = E \\ & \mid \mathsf{Some}(E) \mid op(\overline{v}, E, \overline{e}) \\ & \mid \mathsf{match}\ E\ \mathsf{with}\ \mathsf{Some}(x) \to e_1 \mid \mathsf{None} \to e_2 \\ & \mid \mathsf{match}\ E\ \mathsf{with}\ x_1 :: x_2 \to e_1 \mid \mathsf{nil} \to e_2 \end{aligned}$$

The evaluation proceeds from left to right as denoted by $\overline{v}, E, \overline{e}$ in constructor and dynamic data operation arguments or $v :: E$ in list initialization. We write $e[\overline{x} \leftarrow \overline{v}]$ for the result of replacing variables $\overline{x}$ by values $\overline{v}$ in an expression. The remaining six rules give standard reductions.

Type checking. Well-typed Foo programs reduce to a value in a finite number of steps or get stuck due to an error condition. The stuck states can only be due to the dynamic data operations (e.g. an attempt to convert null value to a number $\mathsf{convFloat}(\mathsf{float}, \mathsf{null})$). The relative safety (Theorem 3) characterizes the additional conditions on input data under which Foo programs do not get stuck.

Typing rules in Figure 7 are written using a judgement $L;\Gamma \vdash e : \tau$ where the context also contains a set of class declarations L. The fragment demonstrates the differences and similarities with Featherweight Java [10] and typing rules for the dynamic data operations *op*:

- All data values d have the type Data, but primitive data values (Booleans, strings, integers and floats) can be implicitly converted to Foo values and so they also have a primitive type as illustrated by the rule for i and f.
- For non-primitive data values (including null, data collections and records), Data is the only type.
- Operations *op* accept Data as one of the arguments and produce a non-Data Foo type. Some of them require a function specifying the conversion for nested values.
- 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 members are well-typed). We consider only classes generated by our type providers and those are well-typed by construction.

4.2 Type providers

So far, we defined the type inference algorithm which produces a shape σ from one or more sample documents (§3) and we defined a simplified model of evaluation of F# (§4.1) and F# Data runtime (§4.2). In this section, 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 preferred shape σ and then use it to generate F# types that are exposed to the programmer.[7]

Type provider mapping. A type provider produces an F# type τ together with a Foo expression and a collection of class definitions. We express it using the following mapping:

$$[\![\sigma]\!] = (\tau, e, L) \qquad (\text{where } L, \emptyset \vdash e : \mathsf{Data} \to \tau)$$

The mapping $[\![\sigma]\!]$ takes an inferred shape σ. It returns an F# type τ and a function that turns the input data (value of type Data) into a Foo value of type τ. The type provider also generates class definitions that may be used by e.

Figure 8 defines $[\![-]\!]$. Primitive types are handled by a single rule that inserts an appropriate conversion function; convPrim just checks that the shape matches and convFloat converts numbers to a floating-point.

[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 or erasure of type provision.

$$[\![\sigma_p]\!] = \tau_p, \lambda x.op(\sigma_p, x), \emptyset \quad \text{where}$$
$$\sigma_p, \tau_p, op \in \{ (\mathsf{bool}, \mathsf{bool}, \mathsf{convPrim}), (\mathsf{int}, \mathsf{int}, \mathsf{convPrim}), (\mathsf{float}, \mathsf{float}, \mathsf{convFloat}), (\mathsf{string}, \mathsf{string}, \mathsf{convPrim}) \}$$

$$[\![\nu\ \{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\}]\!] = C, \lambda x.\mathsf{new}\ C(x), L_1 \cup \ldots \cup L_n \cup \{L\} \quad \text{where}$$
$$C \text{ is a fresh class name}$$
$$L = \mathsf{type}\ C(x_1 : \mathsf{Data}) = M_1 \ldots M_n$$
$$M_i = \mathsf{member}\ \nu_i : \tau_i = \mathsf{convField}(\nu, \nu_i, x_1, e_i),$$
$$\tau_i, e_i, L_i = [\![\sigma_i]\!]$$

$$[\![[\sigma]]\!] = \mathsf{list}\langle\tau\rangle, \lambda x.\mathsf{convElements}(x, e'), L \quad \text{where}$$
$$\tau, e', L = [\![\hat{\sigma}]\!]$$

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

$$[\![\mathsf{nullable}\langle\hat{\sigma}\rangle]\!] = \mathsf{option}\langle\tau\rangle, \lambda x.\mathsf{convNull}(x, e), L \quad \text{where } \tau, e, L = [\![\hat{\sigma}]\!]$$

$$[\![\bot]\!] = [\![\mathsf{null}]\!] = C, \lambda x.\mathsf{new}\ C(x), \{L\} \quad \text{where}$$
$$C \text{ is a fresh class name}$$
$$L = \mathsf{type}\ C(v : \mathsf{Data})$$

Figure 8. Type provider – generation of Foo types from inferred structural types

For records, we generate a class C that takes a data value as a constructor parameter. For each field, we generate a member with the same name as the field. The body of the member calls convField with a function obtained from $[\![\sigma_i]\!]$. This function turns the field value (data of shape σ_i) into a Foo value of type τ_i. The returned expression creates a new instance of C and the mapping returns the class C together with all recursively generated classes. Note that the class name C is not directly accessed by the user and so we can use an arbitrary name, although the actual implementation in F# Data attempts to infer a reasonable name.[8]

A collection shape becomes a Foo list$\langle\tau\rangle$. The returned expression calls convElements (which returns the empty list for data value null). The last parameter is the recursively obtained conversion function for the shape of elements σ. The handling of the nullable shape is similar, but uses convNull.

As discussed earlier, labelled top shapes are also generated as Foo classes with properties. Given any$\langle\sigma_1, \ldots, \sigma_n\rangle$, we get corresponding F# types τ_i and generate n members of type option$\langle\tau_i\rangle$. When the member is accessed, we need to perform a runtime shape test using hasShape to ensure that the value has the right shape (similarly to runtime type conversions from the top type in languages like Java). If the shape matches, a Some value is returned. The shape inference algorithm also guarantees that there is only one case for each shape tag (§3.3) and so we can use the tag for the name of the generated member.

Example 1. To illustrate how the mechanism works, we consider two examples. First, assume that the inferred shape is a record Person { Age : option$\langle$int$\rangle$, Name : string }. The rules from Figure 8 produce the Person class shown below with two members.

The body of the Age member uses convField as specified by the case for optional record fields. The field shape is nullable and so convNull is used in the continuation to convert the value to None if convField produces a null data value and hasShape is used to ensure that the field has the correct shape. The Name value should be always available and should have the right shape so convPrim appears directly in the continuation. This is where the evaluation can get stuck if the field value was missing:

```
type Person(x1 : Data) =
  member Age : option⟨int⟩ =
    convField(Person, Age, x1, λx2 →
      convNull(x2, λx3 → convPrim(int, x3)) )
  member Name : string =
    convField(Person, Name, x1, λx2 →
      convPrim(string, x2)))
```

The function to create the Foo value Person from a data value is $\lambda x.\mathsf{new}\ \mathsf{Person}(x)$.

Example 2. The second example illustrates the handling of collections and labelled top types. Reusing Person from the previous example, consider [any⟨Person {...}, string⟩]:

```
type PersonOrString(x : Data) =
  member Person : option⟨Person⟩ =
    if hasShape(Person {...}, x) then
      Some(new Person(x)) else None
  member String : option⟨string⟩ =
    if hasShape(string, x) then
      Some(convPrim(string, x)) else None
```

The type provider maps the collection of labelled top shapes to a type list⟨PersonOrString⟩ and returns a function that parses a data value as follows:

[8] For example, in {"person":{"name":"Tomas"}}, the nested record will be named Person based on the name of the parent record field.

$\lambda x_1 \rightarrow \mathsf{convElements}(x_1 \lambda x_2 \rightarrow \mathsf{new}\ \mathsf{PersonOrString}(x_2))$

The PersonOrString class contains one member for each of the labels. In the body, they check that the input data value has the correct shape using hasShape. This also implicitly handles null by returning false. As discussed earlier, labelled top types provide easy access to the known cases (string or Person), but they require a runtime shape check.

5. Relative type safety

Informally, the safety property for structural 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 *relative safety*, because we cannot avoid *all* errors. In particular, one can always provide an input that has a different structure than any of the samples. In this case, it is expected that the code will throw an exception in the implementation (or get stuck in our model).

More formally, given a set of sample documents, code using the provided type is guaranteed to work if the inferred shape of the input is preferred with respect to the shape of any of the samples. Going back to §3.2, this means that:

- Input can contain smaller numerical values (e.g., if a sample contains float, the input can contain an integer).
- Records in the input can have additional fields.
- Records in the input can have fewer fields than some of the records in the sample document, provided that the sample also contains records that do not have the field.
- When a labelled top type is inferred from the sample, the actual input can also contain any other value, which implements the open world assumption.

The following lemma states that the provided code (generated in Figure 8) works correctly on an input d' that is a subshape of d. More formally, the provided expression (with input d') can be reduced to a value and, if it is a class, all its members can also be reduced to values.

Lemma 2 (Correctness of provided types)**.** *Given sample data d and an input data value d' such that $S(d') \sqsubseteq S(d)$ and provided type, expression and classes $\tau, e, L = [\![S(d)]\!]$, then $L, e\ d' \rightsquigarrow^* v$ and if τ is a class ($\tau = C$) then for all members N_i of the class C, it holds that $L, (e\ d').N_i \rightsquigarrow^* v$.*

Proof. By induction over the structure of $[\![-]\!]$. For primitives, the conversion functions accept all subshapes. For other cases, analyze the provided code to see that it can work on all subshapes (for example convElements works on null values, convFloat accepts an integer). Finally, for labelled top types, the hasShape operation is used to guaranteed the correct shape at runtime. □

This shows that provided types are correct with respect to the preferred shape relation. Our key theorem states that, for any input which is a subshape the inferred shape and any expression e, a well-typed program that uses the provided types does not "go wrong". Using standard syntactic type safety [26], we prove type preservation (reduction does not change type) and progress (an expression can be reduced).

Theorem 3 (Relative safety)**.** *Assume $d_1, \ldots, d_n$ are samples, $\sigma = S(d_1, \ldots, d_n)$ is an inferred shape and $\tau, e, L = [\![\sigma]\!]$ are a type, expression and class definitions generated by a type provider.*

For all inputs d' such that $S(d') \sqsubseteq \sigma$ and all expressions e' (representing the user code) such that e' does not contain any of the dynamic data operations op and any Data *values as sub-expressions and $L; y : \tau \vdash e' : \tau'$, it is the case that $L, e[y \leftarrow e'\ d'] \rightsquigarrow^* v$ for some value v and also $\emptyset; \vdash v : \tau'$.*

Proof. We discuss the two parts of the proof separately as type preservation (Lemma 4) and progress (Lemma 5). □

Lemma 4 (Preservation)**.** *Given the τ, e, L generated by a type provider as specified in the assumptions of Theorem 3, then if $L, \Gamma \vdash e : \tau$ and $L, e \rightsquigarrow^* e'$ then $\Gamma \vdash e' : \tau$.*

Proof. By induction over $\rightsquigarrow$. The cases for the ML subset of Foo are standard. For (*member*), we check that code generated by type providers in Figure 8 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 Standard ML [15] subset and object-oriented subset [10] of the calculus. The problematic part are the dynamic data operations (Figure 6, Part I). Given a data value (of type Data), the reduction can get stuck if the value does not have a structure required by a specific operation.

The Lemma 2 guarantees that this does not happen inside the provided type. We carefully state that we only consider expressions e' which "[do] not contain primitive operations *op* as sub-expressions". This ensure that only the code generated by a type provider works directly with data values.

Lemma 5 (Progress)**.** *Given the assumptions and definitions from Theorem 3, there exists e'' such that $e'[y \leftarrow e\ d'] \rightsquigarrow e''$.*

Proof. Proceed by induction over the typing derivation of $L; \emptyset \vdash e[y \leftarrow e'\ d'] : \tau'$. The cases for the ML subset are standard. For member access, we rely on Lemma 2. □

6. Practical experience

The F# Data library has been widely adopted by users and is one of the most downloaded F# libraries.[9] A practical demonstration of development using the library can be seen in an attached screencast and additional documentation can be found at http://fsharp.github.io/FSharp.Data.

In this section, we discuss our experience with the safety guarantees provided by the F# Data type providers and other notable aspects of the implementation.

[9] At the time of writing, the library has over 125,000 downloads on NuGet (package repository), 1,844 commits and 44 contributors on GitHub.

6.1 Relative safety in practice

The *relative safety* property does not guarantee safety in the same way as traditional closed-world type safety, but it reflects the reality of programming with external data that is becoming increasingly important [16]. Type providers increase the safety of this kind of programming.

Representative samples. When choosing a representative sample document, the user does not need to provide a sample that represents all possible inputs. They merely need to provide a sample that is representative with respect to data they intend to access. This makes the task of choosing a representative sample easier.

Schema change. Type providers are invoked at compile-time. If the schema changes (so that inputs are no longer related to the shape of the sample used at compile-time), the program can fail at runtime and developers have to handle the exception. The same problem happens when using weakly-typed code with explicit failure cases.

F# Data can help discover such errors earlier. Our first example (§1) points the JSON type provider at a sample using a live URL. This has the advantage that a re-compilation fails when the sample changes, which is an indication that the program needs to be updated to reflect the change.

Richer data sources. In general, XML, CSV and JSON data sources without an explicit schema will necessarily require techniques akin to those we have shown. However, some data sources provide an explicit schema with versioning support. For those, a type provider that adapts automatically could be written, but we leave this for future work.

6.2 Parsing structured data

In our formalization, we treat XML, JSON and CSV uniformly as *data values*. With the addition of names for records (for XML nodes), the definition of structural values is rich enough to capture all three formats.[10] However, parsing real-world data poses a number of practical issues.

Reading CSV data. When reading CSV data, we read each row as an unnamed record and return a collection of rows. One difference between JSON and CSV is that in CSV, the literals have no data types and so we also need to infer the shape of primitive values. 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
```

The value `#N/A` is commonly used to represent missing values in CSV and is treated as null. The Date column uses mixed formats and is inferred as string (we support many date formats and "May 3" would be parsed as date). More interestingly, we also infer Autofiled as Boolean, because the sample contains only 0 and 1. This is handled by adding a bit shape which is preferred of both int and bool.

Reading XML documents. Mapping XML documents to structural values is more interesting. For each node, we create a record. Attributes become record fields and the body becomes a field with a special name. For example:

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

This XML becomes a record root with fields id and • for the body. The nested element contains only the • field with the inner text. As with CSV, we infer shape of primitive values:

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

The XML type provider also includes an option to use *global inference*. In that case, the inference from values (§3.4) unifies the shapes of *all* records with the same name. This is useful because, for example, in XHTML all `<table>` elements will be treated as values of the same type.

6.3 Providing idiomatic F# types

In order to provide types that are easy to use and follow the F# coding guidelines, we perform a number of transformations on the provided types that simplify their structure and use more idiomatic naming of fields. For example, the type provided for the XML document in §6.2 is:

```
type Root =
  member Id : int
  member Item : string
```

To obtain the type signature, we used the type provider as defined in Figure 8 and applied three additional transformations and simplifications:

- When a class C contains a member •, which is a class with further members, the nested members are lifted into the class C. For example, the above type Root directly contains Item rather than containing a member • returning a class with a member Item.
- Remaining members named • in the provided classes (typically of primitive types) are renamed to Value.
- Class members are renamed to follow PascalCase naming convention, when a collision occurs, a number is appended to the end as in PascalCase2. The provided implementation preforms the lookup using the original name.

Our current implementation also adds an additional member to each class that returns the underlying JSON node (called JsonValue) or XML element (called XElement). Those return the standard .NET or F# representation of the value and can be used to dynamically access data not exposed by the type providers, such as textual values inside mixed-content XML elements.

[10] The same mechanism has later been used by the HTML type provider (http://fsharp.github.io/FSharp.Data/HtmlProvider.html), which provides similarly easy access to data in HTML tables and lists.

6.4 Heterogeneous collections

When introducing type providers (§2.3), we mentioned how F# Data handles heterogeneous collections. This allows us to avoid inferring labelled top shapes in many common scenarios. In the earlier example, a sample collection contains a record (with pages field) and a nested collection with values.

Rather than storing a single shape for the collection elements as in $[\sigma]$, heterogeneous collections store multiple possible element shapes together with their *inferred multiplicity* (exactly one, zero or one, zero or more):

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

We omit the details, but finding a preferred common shape of two heterogeneous collections is analogous to the handling of labelled top types. We merge cases with the same tag (by finding their common shape) and calculate their new shared multiplicity (for example, by turning 1 and 1? into 1?).

6.5 Predictability and stability

As discussed in §2, our inference algorithm is designed to be predictable and stable. When a user writes a program using the provided type and then adds another sample (e.g. with more missing values), they should not need to restructure their program. For this reason, we keep the algorithm simple. For example, we do not use probabilistic methods to assess the similarity of record types, because a small change in the sample could cause a large change in the provided types.

We leave a general theory of stability and predictability of type providers to future work, but we formalize a brief observation in this section. Say we write a program using a provided type that is based on a collection of samples. When a new sample is added, the program can be modified to run as before with only small local changes.

For the purpose of this section, assume that the Foo calculus also contains an exn value representing a runtime exception that propagates in the usual way, i.e. $C[\mathsf{exn}] \rightsquigarrow \mathsf{exn}$, and also a conversion function int that turns floating-point number into an integer.

Remark 1 (Stability of inference)**.** *Assume we have a set of samples $d_1, \ldots, d_n$, a provided type based on the samples $\tau_1, e_1, L_1 = [\![S(d_1, \ldots, d_n)]\!]$ and some user code e written using the provided type, such that $L_1; x : \tau_1 \vdash e : \tau$.*

Next, we add a new sample d_{n+1} and consider a new provided type $\tau_2, e_2, L_2 = [\![S(d_1, \ldots, d_n, d_{n+1})]\!]$.

Now there exists e' such that $L_2; x : \tau_2 \vdash e' : \tau$ and if for some d it is the case that $e[x \leftarrow e_1\ d] \rightsquigarrow v$ then also $e'[x \leftarrow e_2\ d] \rightsquigarrow v$.

Such e' is obtained by transforming sub-expressions of e using one of the following translation rules:

1. *$C[e]$ to $C[\mathsf{match}\ e\ \mathsf{with}\ \mathsf{Some}(v) \rightarrow v \mid \mathsf{None} \rightarrow \mathsf{exn}]$*
2. *$C[e]$ to $C[e.\mathsf{M}]$ where $\mathsf{M} = \mathsf{tagof}(\sigma)$ for some σ*
3. *$C[e]$ to $C[\mathsf{int}(e)]$*

Proof. For each case in the type provision (Figure 8) an original shape σ may be replaced by a less preferred shape σ'. The user code can always be transformed to use the newly provided shape:

- Primitive shapes can become nullable (1), int can become float (3) or become a part of a labelled top type (2).
- Record shape fields can change shape (recursively) and record may become a part of a labelled top type (2).
- For list and nullable shapes, the shape of the value may change (we apply the transformations recursively).
- For the any shape, the original code will continue to work (none of the labels is ever removed). □

Intuitively, the first transformation is needed when the new sample makes a type optional. This happens when it contains a null value or a record that does not contain a field that all previous samples have. The second transformation is needed when a shape σ becomes $\mathsf{any}\langle\sigma, \ldots\rangle$ and the third one is needed when int becomes float.

This property also underlines a common way of handling errors when using F# Data type providers. When a program fails on some input, the input can be added as another sample. This makes some fields optional and the code can be updated accordingly, using a variation of (i) that uses an appropriate default value rather than throwing an exception.

7. Related and future work

The F# Data library connects two lines of research that have been previously disconnected. The first is extending the 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# [23, 24], added to Idris [3] and used in areas such as semantic web [18]. The F# Data library has been developed as part of the early F# type provider research, but previous publications focused on the general mechanisms. This paper is novel in that it shows the programming language theory behind a concrete type providers.

Extending the type systems. Several systems integrate external data into a programming language. Those include XML [9, 21] and databases [5]. In both of these, the system 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 [14] is more general, but relies on code generation when importing the schema.

The work that is the most similar to F# Data is the data integration in Cω [13]. It extends C# 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).

In contrast, F# Data type providers do not require any F# language extensions. The simplicity of the Foo calculus shows we have avoided placing strong requirements on the host language. We provide nominal types based on the shapes, rather than adding an advanced system of structural types into the host language.

Advanced type systems and meta-programming. 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 [6, 19] and multi-stage programming [25] could be used to generate code for the provided types; and gradual typing [20, 22] can add typing to existing dynamic languages. 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.

Typing real-world data. Recent work [4] infers a succinct type of large JSON datasets using MapReduce. It fuses similar types based on similarity. This is more sophisticated than our technique, but it makes formal specification of safety (Theorem 3) difficult. Extending our *relative safety* to *probabilistic safety* is an interesting future direction.

The PADS project [7, 11] tackles a more general problem of handling *any* data format. The schema definitions in PADS are similar to our shapes. 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 more challenging.

8. Conclusions

We explored the F# Data type providers for XML, CSV and JSON. As most real-world data does not come with an explicit schema, the library uses *shape inference* that deduces a shape from a set of samples. Our inference algorithm is based on a preferred shape relation. It prefers records to encompass the open world assumption and support developer tooling. The inference algorithm is predictable, which is important as developers need to understand how changing the samples affects the resulting types.

We explored the theory behind type providers. F# Data is a prime example of type providers, but our work demonstrates a more general point. The types generated by type providers can depend on external input and so we can only guarantee *relative safety*, which says that a program is safe only if the actual inputs satisfy additional conditions.

Type providers have been described before, but this paper is novel in that it explores the properties of type providers that represent the "types from data" approach. Our experience suggests that this significantly broadens the applicability of statically typed languages to real-world problems that are often solved by error-prone weakly-typed techniques.

Acknowledgments

We thank to the F# Data contributors on GitHub and other colleagues working on type providers, including Jomo Fisher, Keith Battocchi and Kenji Takeda. We are grateful to anonymous reviewers of the paper for their valuable feedback and to David Walker for shepherding of the paper.

References

[1] L. Cardelli and J. C. Mitchell. Operations on Records. In *Mathematical Foundations of Programming Semantics*, pages 22–52. Springer, 1990.

[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, K. Q. Zhu, and P. White. From Dirt to Shovels: Fully Automatic Tool Generation from Ad Hoc Data. In *Proceedings of ACM Symposium on Principles of Programming Languages*, POPL '08, pages 421–434, 2008. ISBN 978-1-59593-689-9.

[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] H. Mehnert and D. Christiansen. Tool Demonstration: An IDE for Programming and Proving in Idris. In *Proceedings of Vienna Summer of Logic*, VSL'14, 2014.

[13] 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.

[14] 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.

[15] R. Milner. *The Definition of Standard ML: Revised*. MIT press, 1997.

[16] T. Petricek and D. Syme. In the Age of Web: Typed Functional-first Programming Revisited. *Post-Proceedings of ML Workshop*, 2015.

[17] D. Rémy. *Type Inference for Records in a Natural Extension of ML*. Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design. MIT Press, 1993.

[18] 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.

[19] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. In *Proceedings of the ACM Workshop on Haskell*, pages 1–16. ACM, 2002.

[20] J. G. Siek and W. Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[21] 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.

[22] 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.

[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] 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, pages 1–4, 2013.

[25] W. Taha and T. Sheard. Multi-stage Programming with Explicit Annotations. *ACM SIGPLAN Notices*, 32(12):203–217, 1997. ISSN 0362-1340.

[26] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and computation*, 115(1):38–94, 1994.

A. OpenWeatherMap service response

The introduction uses the JsonProvider to access weather information using the OpenWeatherMap service. After registering, you can access the service using a URL http://api.openweathermap.org/data/2.5/weather with query string parameters q and APPID representing the city name and application key. A sample response looks as follows:

```
{
  "coord": {
    "lon": 14.42,
    "lat": 50.09
  },
  "weather": [
    {
      "id": 802,
      "main": "Clouds",
      "description": "scattered clouds",
      "icon": "03d"
    }
  ],
  "base": "cmc stations",
  "main": {
    "temp": 5,
    "pressure": 1010,
    "humidity": 100,
    "temp_min": 5,
    "temp_max": 5
  },
  "wind": { "speed": 1.5, "deg": 150 },
  "clouds": { "all": 32 },
  "dt": 1460700000,
  "sys": {
    "type": 1,
    "id": 5889,
    "message": 0.0033,
    "country": "CZ",
    "sunrise": 1460693287,
    "sunset": 1460743037
  },
  "id": 3067696,
  "name": "Prague",
  "cod": 200
}
```