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

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!

In this paper, we integrate external structured data into the F# type system. 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 and integrate it into the F# type system using type providers.

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

Social network clients, applications for finding tomorrow's weather or searching train schedules all communicate with external services. Increasing number of such services provide REST-based end-points that return data as CSV, XML or JSON. Despite many schematization efforts, most services do not come with an explicit schema. At best, the documentation provides sample responses for typical requests.

For example, http://openweathermap.org/current provides an end-point to get the current weather. The documentation shows one sample to illustrate the typical response. Using standard JSON and web libraries, we might write:


```
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. When the shape is different, the code simply fails with an exception. While not immediately unsound, the code is manifestly 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 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 immensely practical and surprisingly effective in achieving sound information interchange in heterogeneous systems. This paper presents our approach:

- 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 *relativized type safety* for the type providers (§5). This adapts the ML-style type safety for the context of the web.

The supplementary screencast illustrates the practical developer experience using F# Data with JSON, XML and CSV.

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 illustrate the key design principles of the shape inference algorithm:

- The mechanism is predictable. The user directly works with the provided types and should understand why a specific type was produced from a given sample.[1]
- The type providers prefer F# objects with properties. This is to allow extensible (open-world) data formats (§2.2). It also interacts well with developer tooling as most F# editors provide auto-completion on "." and so objects are easier to use than types that require pattern matching.
- Finally, the mechanism handles practical concerns important in the real world. This includes support for different numerical types, null values and missing data.

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       | Null
  | Record of Map<string, JsonValue>
  | Array of JsonValue[]
```

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 approach 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
```

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. 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 generates a type People. Most F# editors also execute the type provider in the background at development-time and use the provided types in auto-completion and 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. The two age values are an integer 25 and a float 3.5 and so the common inferred type is float.

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 what code should be executed at run-time in place of item.Name and other operations. 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 changed.

The role of 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 look at 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 other with Name and Age), but this would complicate the processing code. It is worth noting that languages with stronger tooling around pattern matching such as Idris [11] might have different preferences.

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

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("c:/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 a typed access to elements known 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.[2] This is the important *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 (§3.4). The *labelled top shape* is mapped to the following type:

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

This provides access to the elements known statically from the sample. However the type is deliberately 'weak', because the user needs to explicitly handle the case when a value is not a statically known element. The above code uses Option.iter to skip all unknown elements.

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 (Theorem 2, §6.3).

2.3 Summary

Throughout the introduction, we used data sets that demonstrate the typical problems that are frequent in the real-world (*missing data*, *inconsistent encoding* of primitive values and *heterogeneous shapes*). In our experience, these are the most common issues. The following JSON response with government debt information returned by the World Bank[3] 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": "35.14229" } ] ]
```

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 actual F# Data implementation supports a concept of heterogeneous collections (briefly outlined in §6.3) and provides a type with properties Record for the former and Array for the latter. Second, the value field 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.[4]

In addition to type providers for JSON and XM, 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. 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 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.

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. In the rest of the paper, we assume that record fields can be freely reordered:

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

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

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

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

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

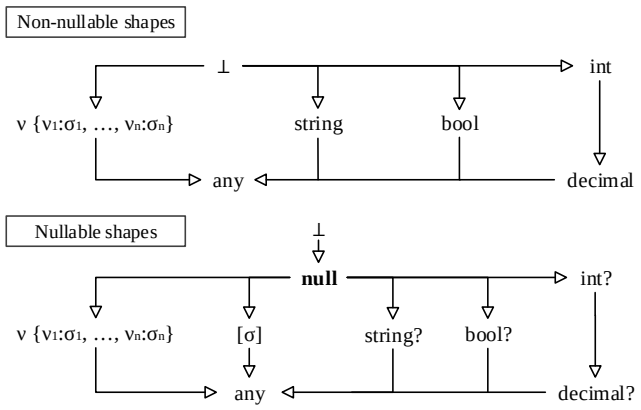

Figure 1. Important aspects of the preferred shape relation

Non-nullable shapes include records (consisting of a name and fields with their shapes) and primitives. Names of records arising from XML are the names of the XML elements while JSON records always use a single name •.

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

Any non-nullable shape $\hat{\sigma}$ can be wrapped as 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. 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 revisit it later by adding labels for statically known alternative shapes (§3.4) as discussed earlier (§2.2).

3.2 Preferred shape relation

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

Definition 1. *We write $\sigma_1 \sqsupseteq \sigma_2$ to denote that σ_2 is preferred over σ_1. The shape preference relation is defined as a transitive reflexive closure of the following rules:*

$$\text{float} \sqsupseteq \text{int} \tag{P1}$$
$$\sigma \sqsupseteq \text{null} \quad (\text{iff } \sigma \neq \hat{\sigma}) \tag{P2}$$
$$\text{nullable}\langle\hat{\sigma}\rangle \sqsupseteq \hat{\sigma} \quad (\text{for all } \hat{\sigma}) \tag{P3}$$
$$\text{nullable}\langle\hat{\sigma_1}\rangle \sqsupseteq \text{nullable}\langle\hat{\sigma_2}\rangle \quad (\text{if } \hat{\sigma_1} \sqsupseteq \hat{\sigma_2}) \tag{P4}$$
$$[\sigma_1] \sqsupseteq [\sigma_2] \quad (\text{if } \sigma_1 \sqsupseteq \sigma_2) \tag{P5}$$
$$\sigma \sqsupseteq \bot \quad (\text{for all } \sigma) \tag{P6}$$
$$\text{any}\langle\sigma_1, \ldots, \sigma_n\rangle \sqsupseteq \sigma \tag{P7}$$

$$\nu\ \{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} \sqsupseteq \nu\ \{\nu_1 : \sigma'_1, .., \nu_n : \sigma'_n\} \quad (\text{if } \sigma_i \sqsupseteq \sigma'_i) \tag{R1}$$

$$\nu\ \{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} \sqsupseteq \nu\ \{\nu_1 : \sigma_1, .., .., \nu_m : \sigma_m\} \quad (\text{when } m \geq n) \tag{R2}$$

$$\nu\ \{\nu_1 : \sigma_1, .., \nu_n : \sigma_n, \nu_{n+1} : \text{null}\} \sqsupseteq \nu\ \{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} \tag{R3}$$

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

- Numeric shape with smaller range is preferred (P1) and we choose 32-bit int over float when possible.
- The null shape is preferred over all nullable shapes (P2), i.e. all shapes excluding non-nullable shapes $\hat{\sigma}$. Any non-nullable shape is preferred over its nullable version (P3); nullable shapes and collections are covariant (P4, P5).
- There is a bottom shape (P6) and any behaves as the top shape, because any shape σ is preferred over any (P7).
- The record shapes are covariant (R1) and preferred record can have additional fields (R2). The rule (R3) implies that records can omit fields, provided that null is a valid value for the field (i.e. the field is optional).

The rule that allows preferred shapes to have fewer fields (R3) is particularly important. This is important, because it allows us to prefer records over the any shape. Given {name : string} and {name : string, age : int}, we find a common shape {name : string, age : int option}. This is a least upper bound and more usable than the top shape var.

Some of the aspects of our system (numeric types and handling of missing data) offer a range of choices. For example, OCaml also has different numerical types, but would likely always handle null explicitly.

3.3 Common preferred shape relation

Given two shapes, the *common preferred shape* relation finds a least upper bound (Theorem 2). It prefers records, which is important for usability as discussed earlier.

Definition 2. *A* common preferred shape *of shapes σ_1 and σ_2 is a shape σ, written $\sigma_1 \triangledown \sigma_2 \vdash \sigma$, obtained according to the inference rules in Figure 2.*

We define shape *tags* to identify shapes that have a common preferred shape which is not the top shape. When finding a common shape of two records (*record*), we return a record that has the union of their fields. The shape of shared fields becomes the common preferred shape of their respective shapes; fields that are present in only one record are marked as nullable using the function $\lceil - \rceil$.

We can find a common shape of two different numbers (*prim*); for two collections, we combine their elements (*list*). When one shape is nullable (*nullable*), we find the common non-nullable shape and wrap it in nullable; if one of the shapes is null, we make the other one nullable (*null*). Finally, if the two shapes do not have matching tags (and they are not nullable), we have to infer the any shape as there is no better alternative. The remaining rules for reflexivity, symmetry and the bottom shape are standard.

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

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

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

$$\text{(nullable)}\ \frac{\hat{\sigma_1} \triangledown \sigma_2 \vdash \sigma}{\mathsf{nullable}\langle\hat{\sigma_1}\rangle \triangledown \sigma_2 \vdash \mathsf{nullable}\langle\lfloor\sigma\rfloor\rangle} \qquad \text{(any)}\ \frac{\mathsf{tagof}(\sigma_1) \neq \mathsf{tagof}(\sigma_2) \qquad \mathsf{tagof}(\sigma_1) \neq \mathsf{nullable} \neq \mathsf{tagof}\sigma_2}{\sigma_1 \triangledown \sigma_2 \vdash \mathsf{any}}$$

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

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

$$\text{(refl)}\ \sigma \triangledown \sigma \vdash \sigma \qquad \text{(null)}\ \sigma \triangledown \mathsf{null} \vdash \lceil\sigma\rceil \quad (\sigma \neq \bot)$$

$$\text{(bot)}\ \bot \triangledown \sigma \vdash \sigma \qquad \text{(prim)}\ \mathsf{int} \triangledown \mathsf{float} \vdash \mathsf{float}$$

Figure 2. Inference judgements that define the common preferred shape relation

Properties. The set of shapes does not have a *unique* least upper bound, but it has a least upper bound with respect to an equivalence between mutually preferred shapes. The common preferred shape relation defines a function (Theorem 1) that finds a least upper bound (Theorem 2).

The equivalence on shapes groups records by their non-nullable fields. Given $\sigma_1 = \mathsf{A}\ \{\mathsf{a} : \mathsf{int}, \mathsf{b} : \mathsf{nullable}\langle\mathsf{int}\rangle\}$, $\sigma_2 = \mathsf{A}\ \{\mathsf{a} : \mathsf{int}\}$ and $\sigma_3 = \mathsf{A}\ \{\mathsf{a} : \mathsf{int}, \mathsf{b} : \mathsf{int}\}$, it is the case that $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_1 \sqsupseteq \sigma_2$ but $\sigma_3 \sqsubseteq \sigma_1$ and not vice versa.

Similarly to labels on the labelled top type (§3.4), the nullable fields of records serve as annotations that provide access to additional values that may or may not be available in the input. In the provided code, these always have to be protected by a runtime check.

Theorem 1 (Preferred shape function)**.** *For all σ_1 and σ_2 there exists exactly one σ such that $\sigma_1 \triangledown \sigma_2 \vdash \sigma$.*

Proof. The pre-conditions of rules in Figure 2 are disjoint, with the exception of (*sym*) and (*refl*), but applying those does not affect the result. Reflexivity is preserved recursively and all rules, symmetry does not enable different derrivations, even for non-symmetric (*nullable*) and (*prim*). □

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

Proof. By induction over $\vdash$. The algorithm only infers the top shape any when for non-nullable shapes of distinct tags and so no better preferred shape exists. □

Next, we extend the core model (sufficient for the relativized safety) with *labelled top types* discussed earleir.

3.4 Adding labelled top type

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

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

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 type, regardless of the labels.

The common preferred shape relation is extended to find a labelled top shape that best represents the sample. We limit the number of labels and avoid nesting by grouping shapes by the shape tag introduced earlier (Figure 2). For example, rather than inferring $\mathsf{any}\langle\mathsf{int}, \mathsf{any}\langle\mathsf{bool}, \mathsf{float}\rangle\rangle$, our algorithm finds the common preferred shape of int and float and produces $\mathsf{any}\langle\mathsf{float}, \mathsf{bool}\rangle$.

The three rules in Figure 3 replace the earlier (*top*). When combining a top with another shape (*top-1*), the labelled top shape may or may not already contain a case with the tag of the other shape. If it does, the two shapes are combined, otherwise a new case is added. When combining two top types (*top-2*), we group the labels that have shared tags. Finally, (*top-3*) covers the case when we are combining two distinct non-top shapes. As top shapes implicitly permit null values, we use an auxiliary function $\lfloor-\rfloor$ to make nullable shapes non-nullable (when possible) to simplify the label.

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

$$\text{(top-2)}\ \frac{(\mathsf{tagof}(\sigma_i) = \mathsf{tagof}(\sigma_j')) \Leftrightarrow (i = j) \wedge (i \leq k) \quad \forall i \in \{1..k\}.(\sigma_i \nabla \sigma_i' \vdash \sigma_i'')}{\mathsf{any}\langle\sigma_1,\ldots,\sigma_k,\ldots,\sigma_n\rangle \nabla \mathsf{any}\langle\sigma_1',\ldots,\sigma_k',\ldots,\sigma_m'\rangle \vdash \mathsf{any}\langle\sigma_1'',\ldots,\sigma_k'',\sigma_{k+1},\ldots,\sigma_n,\sigma_{k+1}',\ldots,\sigma_m'\rangle}$$

$$\text{(top-3)}\ \frac{(\forall i \in \{1,2\}) \quad \mathsf{tagof}(\sigma_1) \neq \mathsf{tagof}(\sigma_2) \quad \mathsf{tagof}(\sigma_i) \neq \mathsf{any} \quad \nexists \sigma_i'.(\sigma_i = \mathsf{option}\langle\sigma_i'\rangle)}{\sigma_1 \nabla \sigma_2 \vdash \mathsf{any}\langle\lfloor\sigma_1\rfloor,\lfloor\sigma_2\rfloor\rangle}$$

Figure 3. Extending the common preferred shape relation for labelled top shapes

Properties. The revised algorithm stil 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 a finer relation than *preferred shape*. In particular, it can only be done with respect to the *sample*. 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.

3.5 Inferring shapes from values

The common preferred shape relation is at the core of the shape inference. What remains to be specified is how we obtain the shape from data. We represent JSON, XML and CSV documents using the same first-order *data* value (§6.2):

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

The first few cases represent primitive values (i for integers, f for floating point numbers and s for strings) and the null value. 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 $\nu_i \mapsto d_i$.

The following defines a mapping $\langle\!\langle d_1,\ldots,d_n\rangle\!\rangle$ which turns a collection of sample data $d_1,\ldots,d_n$ into a shape σ:

$$\begin{aligned} &\langle\!\langle i\rangle\!\rangle = \mathsf{int} \qquad \langle\!\langle \mathsf{null}\rangle\!\rangle = \mathsf{null} \qquad \langle\!\langle \mathsf{true}\rangle\!\rangle = \mathsf{bool} \\ &\langle\!\langle f\rangle\!\rangle = \mathsf{float} \qquad \langle\!\langle s\rangle\!\rangle = \mathsf{string} \qquad \langle\!\langle \mathsf{false}\rangle\!\rangle = \mathsf{bool} \\ &\langle\!\langle d_1,\ldots,d_n\rangle\!\rangle = \sigma_n \quad \text{where} \\ &\quad \sigma_0 = \bot,\ \forall i \in \{1..n\}.\ \sigma_{i-1} \nabla \langle\!\langle d_i\rangle\!\rangle \vdash \sigma_i \\ &\langle\!\langle [d_1;\ldots;d_n]\rangle\!\rangle = [\langle\!\langle d_1,\ldots,d_n\rangle\!\rangle] \\ &\langle\!\langle \nu\,\{\nu_1 \mapsto d_1,\ldots,\nu_n \mapsto d_n\}\rangle\!\rangle = \\ &\quad \nu\,\{\nu_1 : \langle\!\langle d_1\rangle\!\rangle,\ldots,\nu_n : \langle\!\langle d_n\rangle\!\rangle\} \end{aligned}$$

We overload the notation and write $\langle\!\langle s\rangle\!\rangle$ when inferring shape from a single value. Primitive values are mapped to their corresponding shapes. For records, we return infer field shapes from the individual values.

When infering 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 at the top-level (when calling a type provider with multiple samples) and also when inferring the shape of collection elements.

4. Formalising type providers

This section presents the theoretical model of strucutral type providers. We introduce the Foo calculus which is an object-oriented subset of F# based on SML [14] and Featherweight Java [9]. It includes operations for working with data that model the F# Data runtime. Finally, we describe how type providers turn inferred shapes into Foo classes (§4.2).

Type providers map the dirty world of data into a nice world of F#. To model this, the Foo calculus does not have null values and data values d are never directly exposed.

4.1 The Foo calculus

The syntax of the calculus is shown in Figure 5. 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.

The type Data is the type of all structural data d. A class definition L consists of a constructor and zero or more members. Values v include previously defined data d; expressions e include class construction, member access and the usual functional constructs (functions, lists, options) and conditionals. The op constructs are discussed next.

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

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

Figure 5. The syntax of the Foo calculus

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

$$\mathsf{hasShape}([\sigma],[d_1;\ldots;d_n]) \rightsquigarrow \mathsf{hasShape}(\sigma,d_1)\wedge\ldots\wedge\mathsf{hasShape}(\sigma,d_n)$$
$$\mathsf{hasShape}([\sigma],\mathsf{null}) \rightsquigarrow \mathsf{true}$$
$$\mathsf{hasShape}(\mathsf{string},s) \rightsquigarrow \mathsf{true}$$
$$\mathsf{hasShape}(\mathsf{int},i) \rightsquigarrow \mathsf{true}$$
$$\mathsf{hasShape}(\mathsf{bool},d) \rightsquigarrow \mathsf{true} \quad (\text{when } d \in \mathsf{true},\mathsf{false})$$
$$\mathsf{hasShape}(\mathsf{float},d) \rightsquigarrow \mathsf{true} \quad (\text{when } d=i \text{ or } d=f)$$
$$\mathsf{hasShape}(\_,\_) \rightsquigarrow \mathsf{false}$$

$$\mathsf{convFloat}(\mathsf{float},i) \rightsquigarrow f\ (f=i)$$
$$\mathsf{convFloat}(\mathsf{float},f) \rightsquigarrow f$$
$$\mathsf{convNull}(\mathsf{null},e) \rightsquigarrow \mathsf{None}$$
$$\mathsf{convNull}(d,e) \rightsquigarrow \mathsf{Some}(e\ d)$$
$$\mathsf{convPrim}(\sigma,d) \rightsquigarrow d \quad (\sigma,d \in \{(\mathsf{int},i),(\mathsf{string},s),(\mathsf{bool},b)\})$$
$$\mathsf{convField}(\nu,\nu_i,\nu\ \{\ldots,\nu_i=d_i,\ldots\},e) \rightsquigarrow e\ d_i$$
$$\mathsf{convField}(\nu,\nu',\nu\ \{\ldots,\nu_i=d_i,\ldots\},e) \rightsquigarrow e\ \mathsf{null} \quad (\nexists i.\nu_i=\nu')$$
$$\mathsf{convElements}([d_1;\ldots;d_n],e) \rightsquigarrow e\ d_1 :: \ldots :: e\ d_n :: \mathsf{nil}$$
$$\mathsf{convElements}(\mathsf{null}) \rightsquigarrow \mathsf{nil}$$

Figure 4. Reduction rules for conversion functions

The behaviour of the dynamic data operations is defined by the reduction rules in Figure 4. The operations can convert data values into values of less preferred shape. For example, convFloat turns an integer 1 into a floating-point 1.0 and convElements turns null value into an empty list.

All of the dynamic data operations take a data value d and produce a Foo value. We do not make the conversion explicit for primitive values (convPrim) and assume that primitive values are shared, but we convert all other values. For example convElements turns a data collection $[d_1,\ldots,d_n]$ into a Foo list $v_1 :: .. :: v_n :: \mathsf{nil}$.

The data operations for collections, record fields and nullable values all take a function as the last argument and invoke it to convert the obtained data. For example, convField accesses a record field – it ensures that the actual record name matches the expected name and passes the field value (or null data value) to the provided function.

Finally, we also define a runtime shape test hasShape. The most complex case is handling of records where we 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. Although not elegant, if we treat $e_1 \vee e_2$ and $e_1 \wedge e_2$ as syntactic sugar for if then the rule is expressed using just Foo expressions.

Reduction. The reduction relation is of the form $L, e \rightsquigarrow e'$. We often omit class declarations L and we write $L, e \rightsquigarrow^* e'$ to denote the reflexive and transitive closure of $\rightsquigarrow$.

Figure 6 shows the reduction rules. The (*ctx*) rule models the eager evaluation of F# and performs a reduction inside a sub-expression specified by an evaluation context E:

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

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 (*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 a functional language.

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. $\mathsf{convFloat}(\mathsf{float},\mathsf{null})$). The relativized type safety (Theorem 4) characterizes the additional conditions on input data under which Foo programs do not "go wrong".

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 SML and Featherweight Java:

- 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* (omitted) have a family of types (Doo does not have polymorphic types), accepting Data as one of the arguments and producing a non-data Foo 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 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.

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

$$\text{(match1)}\quad \begin{array}{l}\text{match None with}\\ \text{Some}(x) \to e_1 \mid \text{None} \to e_2\end{array} \rightsquigarrow\ e_2$$

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

$$\text{(match3)}\quad \begin{array}{l}\text{match nil with}\\ x_1 :: x_2 \to e_1 \mid \text{nil} \to e_2\end{array} \rightsquigarrow\ e_2$$

$$\text{(match4)}\quad \begin{array}{l}\text{match } v_1 :: v_2 \text{ with}\\ x_1 :: x_2 \to e_1 \mid \text{nil} \to e_2\end{array} \rightsquigarrow\ e_1[\overline{x} \leftarrow \overline{v}]$$

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

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

$$\text{(eq1)}\quad v = v \rightsquigarrow \text{true} \qquad \text{(eq2)}\quad v = v' \rightsquigarrow \text{false}$$

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

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

Figure 6. Featherweight F# – Remaining reduction rules

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.[5]

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:

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

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.

For records, we generate a class C that takes a data value as constructor parameter. For each record field, we generate a member with the same name as the field.[6] 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 arbitrary name, although the actual implementation in F# Data attempts to infer a reasonable name.[7]

$$\frac{}{L;\Gamma \vdash d : \text{Data}} \qquad \frac{}{L;\Gamma \vdash i : \text{int}} \qquad \frac{}{L;\Gamma \vdash f : \text{float}}$$

$$\frac{L;\Gamma, x : \tau_1 \vdash e : \tau_2}{L;\Gamma \vdash \lambda x.e : \tau_2} \qquad \frac{L;\Gamma \vdash e_2 : \tau_1 \quad L;\Gamma \vdash e_1 : \tau_1 \to \tau_2}{L;\Gamma \vdash e_1\ e_2 : \tau_2}$$

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

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

Figure 7. Featherweight F# – Fragment of type checking

A collection shape becomes an Foo list$\langle\tau\rangle$. The returned expression calls convElements (which returns 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 shape 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 type is a record { Age : option$\langle$int$\rangle$, Name : string }. The rules from Figure 8 produce 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 and wraps the result in Some. Note that getField is defined

[5] The actual implementation provides *erased types* as described in [21]. 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.

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

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

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

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

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

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

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

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

Figure 8. Type provider – generation of featherweight F# types from inferred structural types

even when the field does not exist, but it returns null. This lets us treat missing fields as optional. An F# type corresponding to the (structural) record is Person and a structural value s is wrapped by by calling new Person(s).

Example 2. The second example illustrates the remaining types including collections and variants. Reusing Person from the previous example, consider [any⟨Person, string⟩], a collection which contains a mix of Person and string values:

```
type PersonOrString(v : StructVal) =
  member Person : option⟨Person⟩ =
    if hasShape({Age:option⟨int⟩, Name:string}, v)
      then Some(new Person(v)) else None
  member String : option⟨string⟩ =
    if hasShape(string, v)
      then Some(asStr(v)) else None
```

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

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

The PersonOrString type contains one member for each of the variant case. In the body, they check that the value has the correct type using hasShape. This also implicitly handles null by returning false. As discussed earlier, variants provide easy access to the known cases (string or Person), but they are robust with respect to unexpected inputs.

5. Relativized 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 *relativized 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 type of the input is a subtype 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, provided that the type of the field is missing in some of the samples
- When a variant is inferred from the sample, the actual input can also contain any other value

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

Lemma 3 (Correctness of provided types)**.** *Given a sample value s and an input value s' such that $\langle\!\langle s \rangle\!\rangle \sqsupseteq \langle\!\langle s' \rangle\!\rangle$ 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 $[\![-]\!]_-$. For primitives, the conversion functions accept all subtypes. For other cases, analyze 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, for variants, the value has the correct type as guaranteed by hasShape. □

This shows that the provided types are correct with respect to subtyping. Our key theorem 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 standard syntactic type safety [24], we prove type preservation (reduction does not change type) and progress (an expression that is not a value can be reduced).

Theorem 4 (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.*

For all new inputs s' such that $\exists i.(\langle\!\langle s_i \rangle\!\rangle \sqsupseteq \langle\!\langle s' \rangle\!\rangle)$, 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 e_c does not contain primitive operations op as a sub-expression, 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 5) and progress (Lemma 6). □

Lemma 5 (Preservation)**.** *Given the class definitions L generated by a type provider as specified in the assumptions of Theorem 4, 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 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 subset and object-oriented subset of the calculus. The problematic part are the primitive functions (Figure 4). Given a structural value (of type StructVal), the reduction can get stuck if the value does not have a structure required by a specific primitive.

The Lemma 3 guarantees that this does not happen inside the provided type. In Theorem 4, we carefully state that we only consider expressions e_c which "[do] not contain primitive operations *op* as sub-expressions". This makes sure that the only code working with structural values is the code generated by the type provider.

Lemma 6 (Progress)**.** *Given the assumptions and definitions from Theorem 4, it is the case that $e_c[y \leftarrow e_s] \rightsquigarrow e'_c$.*

Proof. Proceed 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 3. □

6. Practical experience

The F# Data library has been widely adopted by the industry and is one of the most downloaded F# libraries.[8] 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 practical experience with the safety guarantees provided by the F# Data type providers and other notable aspects of the implementation.

6.1 Relativized safety in practice

The *relativized safety* property does not guarantee the same amount of safety as traditional ML type safety, but it reflects the reality of programming with external data sources that is increasingly important [15]. Type providers do not reduce the safety – they simply illuminate the existing issues.

One such issue is the handling of schema change. With type providers, the sample is captured at compile-time. If the schema changes later (so that the input is no longer a subtype of the sample), the program fails at runtime and developers have to handle the exception. This is the same problem that happens when reading data using any other library.

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 schema changes, which is an indication that the program needs to be updated to reflect the change.

In general, there is no better solution for plain XML, CSV and JSON data sources. However, 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 work.

6.2 Parsing structured data

In our formalization, we treat XML, JSON and CSV uniformly as *structural values*. With the addition of named records (for XML nodes), the definition of structural values is rich enough to capture all three formats[9]. 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 type 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 correctly). More interestingly, we also infer Autofiled as Boolean, because the sample contains only 0 and 1. This is handled by adding a bit type which is a subtype of both int and bool.

Reading XML documents. Mapping XML documents to structural values is perhaps the most interesting. For each

[8] At the time of writing, the library has over 80,000 downloads on NuGet (package repository), 1,821 commits and 44 contributors on GitHub.

[9] 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.

node, we create a named record. Attributes become record fields and the body becomes a field with a special name:

```
<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 type of primitive values:

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

When generating types for XML documents, we also lifts the members nested under the • field into the parent type to simplify the structure of the generated type.

The XML type provider also includes an option to use *global inference*. In that case, the inference from values (3.5) unifies the types 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 Heterogeneous collections

When introducing type providers (§2.3), we mentioned that F# Data implements a special handling of heterogeneous collections. This allows us to avoid inferring a variant types in many common scenarios. In the earlier example, a sample collection contains a record (with pages and page fields) and a nested collection with values.

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

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

We omit the details, but finding a common supertype of two heterogeneous collections is analogous to the handling of variants. We merge cases with the same tag (by finding their common super-type) and calculate their new shared multiplicity (for example, by turning 1 and 1? into 1?).

7. Related and future work

We connect two lines of research: integration of external data into statically-typed programming languages and inferring types for real-world data sources. We build on F# type providers [2, 16, 21, 22], but our paper is novel in that it discusses the theory and safety of a concrete type provider.

Extending the type systems. Previous work that integrates external data, such as XML [8, 19] and databases [4, 13], into a programming language, required the user to define the schema or it has an ad-hoc extension that reads the schema.

Cω [12] is the most similar to F# Data. It extends C# with types similar to our structural types (including similar heterogeneous collections), but it does not infer the types from a sample and extends the type system of the host language (rather than using general purpose embedding).

Advanced type systems. A number of other advanced type system features could be used to tackle the problem discussed in this paper. The Ur [1] language has a rich system for working with records; meta-programming [17], [5] and multi-stage programming [23] could be used to generate code for the provided types; and gradual typing [18, 20] 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 [3] 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 4) difficult. Extending our *relativized safety* to *probabilistic safety* is an interesting future work.

The PADS project [6, 10] 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 [7] 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

We explored the F# Data type providers for 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. Our type inference algorithm is based on a common supertype relation. For usability reasons, it prefers records and avoids variant types. The algorithm is predictable, which is important as developers need to understand how changing the samples affects the resulting types.

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 and so we can only guarantee *relativized safety*, 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.

Type providers have 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 for F# Data type providers, which have been widely adopted by the industry.

References

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

[2] 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.

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

[4] 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.

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

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[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] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[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, pages 1–4, 2013. doi: 10.1145/2429376.2429378.

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

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