F# Data: Making structured data first-class citizens

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 do not come with an explicit schema, we develop a type inference algorithm that infers a type from representative sample documents and integrate it into the F# type system using type providers.

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

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

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

1. Introduction

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

For example, OpenWeatherMap provides an end-point for getting current weather for a given city[1]. The documentation 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 format described in the documentation. The root node must be a record with a `"main"` field, which has to be another record containing a numerical `"temp"` field. When the format is incorrect, the data access simply fails with an exception.

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

```
type W = JsonProvider⟨"http://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.

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

- We present F# Data type providers for XML, CSV and JSON (§2) and practical aspects of their implementation that contributed to their industrial adoption (§6).
- We describe a predictable type inference algorithm for structured data formats, based on a *preferred typing* 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 show developer experience using F# Data with JSON, XML and CSV.

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

2. Structural type providers

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

- Our mechanism is predictable. The user directly works with the inferred types and should understand why a specific type was inferred from a given sample.[2]
- Our inference mechanism prefers records. This supports developer tooling – most F# editors provide code completion hints on "." and so types with properties (records) are easier to use than types that require pattern matching.
- Finally, we handle a number of practical concerns 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 used in §1 is a popular data exchange format based on JavaScript data structures. The following is the definition of JsonValue used earlier to represent JSON:

```
type JsonValue =
  | Number of decimal   | 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 } ]
```

Say we want to print the names and an age if it is available. The standard approach 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 in an incorrect format. Even with helper functions, we would still need to 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 in the same format, 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 the same 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.

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

```
type Entity =
  member Name : string
  member Age : option<decimal>

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

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

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 is in the same format 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 the document does not have the expected format.

Informally, the safety property (§5) states that if the inputs are subtypes of some of the provided samples (i.e. the samples are representative), then no exceptions will occur. In other words, we cannot avoid all failures, but we can prevent some. Moreover, if the OpenWeatherMap changes the response format, the sample in §1 will not re-compile and the user knows that the code needs to be changed.

The role of records. The sample code is easy to write thanks to the fact that most F# editors provide code completion when "." is typed (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 inference algorithm prefers records (we also treat XML elements and CSV rows as records) and only infers types that require pattern matching as the last resort.

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

[2] 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 element, one of which is `<heading>` representing headings:

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

The F# Data library has been designed primarilly 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 element 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 and variant types. We do not infer a union type representing a choice between heading, paragraph and image. By its nature, XML is extensible and the sample cannot include all possible nodes.[3] The type of elem is thus a top type (we call it a *variant*) annotated with the statically known possibilities. It is mapped to the following F# type:

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

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

As a consequence, the variant type can be seen as the top type. The above code will work correctly on document that contains any elements, including those not listed in the sample. This is extremely useful when reading data from external sources, because the sample document only needs to contain values relevant to the application. It is also worth noting that our type inference attempts to minimze the number of variant types that appear in the resulting type – a variant is not used if there is another option (see Corollary 3 and §6.3).

2.3 Summary

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.

Throughout the introduction, we used data sets that demonstrate typical issues that are frequent in the real-world (missing data, inconsistent encoding of primitive values and heterogeneous structures). In our experience, these are the most common issues. The following JSON response with government debt information returned by the World Bank API[4] 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.[5]

3. Inferring Types from Data

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

We first define the type of structured data σ. Note that this type is distinct from the programming language types τ (type providers generate the latter from the former). Next, we define the subtyping relation on structural types σ and describe the algorithm for finding a preferred typing.

3.1 Inferred Types

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

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

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

[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.

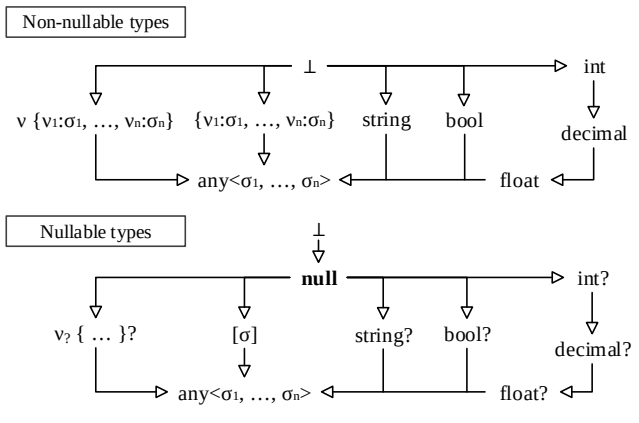

Figure 1. Preference relation between types inferred from data samples

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

The three numerical primitives, int for integers, decimal for small precise decimal numbers and float for floating-point numbers, are related in a preference order as discussed in §3.2.

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

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

3.2 Preference relation

Figure 1 provides a basic intuition about the relationship between possible types inferred from data samples. The upper part shows non-nullable types (with records and primitive types) and the lower part shows nullable types with null, collections and optional values. We abbreviate $\mathsf{option}\langle\sigma\rangle$ as $\sigma?$ and the diagram omits links between the two parts; any type $\hat{\sigma}$ is preferred to $\mathsf{option}\langle\hat{\sigma}\rangle$.

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

$$\mathsf{float} \leq \mathsf{decimal} \leq \mathsf{int} \quad \text{(P1)}$$

$$\sigma \leq \mathsf{null} \quad (\text{iff } \sigma \neq \hat{\sigma}) \quad \text{(P2)}$$

$$\mathsf{option}\langle\hat{\sigma}\rangle \leq \hat{\sigma} \quad (\text{for all } \hat{\sigma}) \quad \text{(P3)}$$

$$\mathsf{option}\langle\hat{\sigma_1}\rangle \leq \mathsf{option}\langle\hat{\sigma_2}\rangle \quad (\text{if } \hat{\sigma_1} \leq \hat{\sigma_2}) \quad \text{(P4)}$$

$$[\sigma_1] \leq [\sigma_2] \quad (\text{if } \sigma_1 \leq \sigma_2) \quad \text{(P5)}$$

$$\sigma \leq \bot \quad (\text{for all } \sigma) \quad \text{(P6)}$$

$$\mathsf{any}\langle\sigma_1, \ldots, \sigma_n\rangle \leq \sigma \quad \text{(P7)}$$

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

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

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

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

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

- Numeric types (P1) are ordered by the range they represent; int is a 32-bit integer, decimal has a range of -1^{29} to 1^{29} and float is a floating-point number (but imprecise).
- The null type preferred to all nullable types (P2), i.e. all types excluding non-nullable types $\hat{\sigma}$. Any non-nullable type is a preferred to its nullable version (P3) and both options and collections are covariant (P4, P5).
- There is a bottom type (P6) and variants behave as top types, because any type σ is a subtype of any variant (P7). There is a range of variants (no *single* top type), but any variant is a subtype of any other variant.
- As usual, the subtyping on records is covariant (R1), subtype can have additional fields (R2) and fields can be reordered (R3). The interesting rule is (R4). It implies that subtype can omit fields, provided that null is a valid value for the field (i.e. the field is optional).

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

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

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

$$\begin{array}{rcl}
\mathsf{tag} & = & \mathsf{bool} \\
& | & \mathsf{string} \quad | \quad \mathsf{number} \\
& | & \mathsf{any} \quad | \quad \mathsf{collection} \\
& | & \mathsf{record} \quad | \quad \mathsf{named}\ \nu
\end{array}$$

$$\mathsf{tagof}(\mathsf{string}) = \mathsf{string} \qquad \mathsf{tagof}(\mathsf{bool}) = \mathsf{bool}$$
$$\mathsf{tagof}([\sigma]) = \mathsf{collection} \qquad \mathsf{tagof}(\hat{\sigma}\ \mathsf{option}) = \mathsf{tagof}(\hat{\sigma})$$

$$\mathsf{tagof}(\mathsf{any}\langle\sigma_1, \ldots, \sigma_n\rangle) = \mathsf{any}$$
$$\mathsf{tagof}(\{\nu_1 : \sigma_1,\ \ldots\ , \nu_n : \sigma_n\}) = \text{rec-anon}$$
$$\mathsf{tagof}(\nu\ \{\nu_1 : \sigma_1,\ \ldots\ , \nu_n : \sigma_n\}) = \text{rec-named}\ \nu$$
$$\mathsf{tagof}(\sigma) = \mathsf{number} \quad \sigma \in \{\mathsf{int}, \mathsf{decimal}, \mathsf{float}\}$$

$$\lceil \hat{\sigma} \rceil = \hat{\sigma}\ \mathsf{option} \quad \text{(non-nullable types)} \qquad \lfloor \hat{\sigma}\ \mathsf{option} \rfloor = \hat{\sigma} \quad \text{(option)}$$
$$\lceil \sigma \rceil = \sigma \quad \text{(otherwise)} \qquad \lfloor \sigma \rfloor = \sigma \quad \text{(otherwise)}$$

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

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

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

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

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

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

$$\text{(refl)}\ \sigma \triangledown \sigma \vdash \sigma \qquad \text{(null-1)}\ \sigma \triangledown \mathsf{null} \vdash \sigma \quad (\sigma \leq \mathsf{null})$$
$$\text{(bot)}\ \bot \triangledown \sigma \vdash \sigma \qquad \text{(null-2)}\ \sigma \triangledown \mathsf{null} \vdash \sigma\ \mathsf{option} \quad (\sigma :\not\geq \mathsf{null})$$

Figure 2. Inference judgements that define the most preferred typing relation

3.3 Most preferred typing relation

Given two typings, the *most preferred typing* relation finds a greatest lower bound (Theorem 2). When possible, it avoids variant types and prefers records (Corollary 3), which is important for usability as discussed earlier (§2.1).

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

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

Although all variants are mutual subtypes, we find one that best represents the sample. We avoid nested variants and limit the number of cases. This is done by grouping types that have a preferred typing which is not a variant. For example, rather than inferring $\mathsf{any}\langle\mathsf{int}, \mathsf{any}\langle\mathsf{bool}, \mathsf{decimal}\rangle\rangle$, our algorithm finds the preferred typing of int and decimal and produces $\mathsf{any}\langle\mathsf{decimal}, \mathsf{bool}\rangle$. To identify types that have a preferred typing which is not a variant, we group the types by a tag. The tag of a type is obtained using a function tagof(-). The function is not defined on $\bot$ and null as those are hanled by the (*bot*) and (*null-1*) or (*null-2*).

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

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

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

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

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

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

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

$$\mathsf{hasType}(\nu_?\{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\}, \nu'_?\{\nu'_1 \mapsto s_1, \ldots, \nu'_m \mapsto s_m\}) \rightsquigarrow (\nu_? = \nu'_?) \wedge$$
$$(\ ((\nu_1 = \nu'_1) \wedge \mathsf{hasType}(\sigma_1, s_1)) \vee \ldots \vee ((\nu_1 = \nu'_m) \wedge \mathsf{hasType}(\sigma_1, s_m)) \vee \ldots \vee$$
$$((\nu_n = \nu'_1) \wedge \mathsf{hasType}(\sigma_n, s_1)) \vee \ldots \vee ((\nu_n = \nu'_m) \wedge \mathsf{hasType}(\sigma_n, s_m))\ )$$

$$\mathsf{hasType}([\sigma], [s_1; \ldots; s_n]) \rightsquigarrow \mathsf{hasType}(\sigma, s_1) \wedge \ldots \wedge \mathsf{hasType}(\sigma, s_n)$$
$$\mathsf{hasType}([\sigma], \mathsf{null}) \rightsquigarrow \mathsf{true}$$

$$\mathsf{hasType}(\mathsf{string}, t) \rightsquigarrow \mathsf{true}$$
$$\mathsf{hasType}(\mathsf{bool}, s) \rightsquigarrow \mathsf{true} \quad (\text{when } s \in \mathsf{true}, \mathsf{false})$$
$$\mathsf{hasType}(\mathsf{float}, s) \rightsquigarrow \mathsf{true} \quad (\text{when } s = i, s = d \text{ or } s = f)$$
$$\mathsf{hasType}(\mathsf{decimal}, s) \rightsquigarrow \mathsf{true} \quad (\text{when } s = i \text{ or } s = f)$$
$$\mathsf{hasType}(\mathsf{int}, i) \rightsquigarrow \mathsf{true}$$
$$\mathsf{hasType}(\mathsf{option}\langle\sigma\rangle, s) \rightsquigarrow \mathsf{hasType}(\sigma, s)$$
$$\mathsf{hasType}(\mathsf{option}\langle\sigma\rangle, s) \rightsquigarrow \mathsf{true}$$
$$\mathsf{hasType}(\mathsf{any}\langle\ldots\rangle, s) \rightsquigarrow \mathsf{true}$$
$$\mathsf{hasType}(\_, \_) \rightsquigarrow \mathsf{false}$$

Figure 3. Reduction rules for conversion functions

Properties. The partially ordered set of types does not have a *unique* least upper bound, but it does have a least upper bound with respect to an equivalence between mutual subtypes. The common supertype relation is well-defined (Theorem 1) and finds the least upper bound (Theorem 2).

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

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

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

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

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

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

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

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

3.4 Inferring types from values

The preferred typing algorithm (§3) is at the core of the type inference for structural values, but we have not yet clarified how it is used. Given a JSON, XML or CSV document, F# Data constructs a structural value s from the sample. The following defines a mapping $\langle\!\langle s_1, \ldots, s_n \rangle\!\rangle$ which turns a collection of sample values $s_1, \ldots, s_n$ into a structural type σ:

$$\langle\!\langle i \rangle\!\rangle = \mathsf{int} \qquad \langle\!\langle d \rangle\!\rangle = \mathsf{decimal} \qquad \langle\!\langle b \rangle\!\rangle = \mathsf{bool}$$
$$\langle\!\langle f \rangle\!\rangle = \mathsf{float} \qquad \langle\!\langle t \rangle\!\rangle = \mathsf{string} \qquad \langle\!\langle \mathsf{null} \rangle\!\rangle = \mathsf{null}$$

$$\langle\!\langle s_1, \ldots, s_n \rangle\!\rangle = \sigma_n \quad \text{where}$$
$$\sigma_0 = \bot,\ \forall i \in \{1..n\}.\ \sigma_{i-1} \triangledown \langle\!\langle s_i \rangle\!\rangle \vdash \sigma_i$$
$$\langle\!\langle [s_1; \ldots; s_n] \rangle\!\rangle = [\langle\!\langle s_1, \ldots, s_n \rangle\!\rangle]$$
$$\langle\!\langle \nu?\ \{\nu_1 \mapsto s_1, \ldots, \nu_n \mapsto s_n\} \rangle\!\rangle =$$
$$\nu?\ \{\nu_1 : \langle\!\langle s_1 \rangle\!\rangle, \ldots, \nu_n : \langle\!\langle s_n \rangle\!\rangle\}$$

We overload the notation and write $\langle\!\langle s \rangle\!\rangle$ when inferring type from a single value. Primitive values are mapped to their corresponding types. For records, we return a type with field types inferred based on individual values.

When infering a value from multiple samples, we use the preferred typing relation to find a common type 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 type of a collection.

4. Formalising type providers

In this section, we build the theoretical framework for proving relativised type safety of structural type providers. We discuss the runtime representation of documents and primitive operations (§4.1), we embed these into a formal model of an F# subset (§4.2) and we describe how structural type providers turn inferred structural types into F# types (§4.3).

4.1 Structural values and conversions

We represent JSON, XML and CSV documents using the same first-order *structural values* defined as follows:

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

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

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

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

The operations are used in code generated by type providers. Their behavior is defined by the rules in Figure 3. The conversion functions (asDec and asFloat) only perform conversions required by the subtyping (our actual implementation is more lax and performs additional conversions).

The getField operation is used to access a field of a record. The operation ensures that the actual record name matches the expected name (we write $\bullet$ for the name of an unnamed record). The getChildren operation returns elements of a list and turns null into an empty collction.

Finally, we also define two helper functions. The isNull operation is used to check whether value is null and hasType is a runtime type test. The most complex case is handling of records where we check that for each field $\nu_1, \ldots, \nu_n$ in the type, the actual record value has a field of the same name with a matching type. The last line defines a "catch all" pattern, which returns false for all remaining cases. Although not elegant, the rules can be expressed using just Featherweight F#, discussed in the next section.

4.2 Featherweight F#

We now extend the semantics discussed so far and add a minimal subset of F# needed to model F# Data. Type providers use classes and so we focus on the F# object model and combine aspects of SML [**?**] and Featherweight Java [**?**].

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

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

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

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

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

Reduction. The reduction relation is of the form $e \rightsquigarrow e'$. We also write $e \rightsquigarrow^* e'$ to denote the reflexive and transitive closure of $\rightsquigarrow$. The reduction rules for operations op were discussed earlier. Figure 4 shows the remaining rules.

The (*ctx*) rule performs a reduction inside a sub-expression specified by an evaluation context. This models the eager

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

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

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

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

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

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

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

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

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

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

Figure 4. Featherweight F# – Remaining reduction rules

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

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

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

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

Figure 5. Featherweight F# – Fragment of type checking

evaluation of F#. An evaluation context E is defined as:

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

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

We write $e[\overline{x} \leftarrow \overline{v}]$ for the result of replacing variables $\overline{x}$ by values $\overline{v}$ in 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 conditionals and pattern matching.

We omit expressions $e \vee e$ and $e \wedge e$ that are used in Figure 3, as those can be easily defined as syntactic sugar using if. All expressions reduce to a value in a finite number of steps or get stuck due to an error condition. An error condition can be a wrong argument passed to conditional, pattern matching or one of the primitives from Figure 3.

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

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

An important part of Featherweight Java that is omitted here is the checking of type declarations (ensuring the members are well-typed). We consider only classes generated by our type providers and those are well-typed by construction.

4.3 Type providers

So far, we defined the type inference algorithm which produces a structural type σ from one or more sample documents (§3) and we defined a simplified model of evaluation of F# (§4.2) and F# Data runtime (§4.3). 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 preferred typing σ and then use it to generate F# types that are exposed to the programmer.[6]

Type provider mapping. A type provider produces an F# type τ together with an expression that wraps an input value (of type Data) as a value of type τ and a collection of class definitions. We express it using the following mapping:

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

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

Figure 6 shows the rules that define $[\![-]\!]_-$. Primitive types are all handled by two rules – for decimal and float, we insert a call to an appropriate conversion function from Figure 3.

For records, we generate a class C that takes a structural value as constructor parameter. For each record field, we generate a member with the same name as the field.[7] The body of the member calls getField and then wraps this expression using $[\![\sigma_i]\!]$ which maps the field (structural value of type σ_i) into an F# 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.[8]

A collection type becomes an F# $\mathsf{list}\langle\tau\rangle$. The returned expression calls getChildren (which turns null into an empty list) and then uses map to convert nested values to values of an F# type τ. The handling of option type is similar; if the value is not null, we wrap the recursively generated conversion expression e' in the Some constructor.

As discussed earlier, variant types are also generated as F# classes with properties. Given a variant $\mathsf{any}\langle\sigma_1, \ldots, \sigma_n\rangle$,

[6] The actual implementation provides *erased types* as described in [**?**]. 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.

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

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

$$\begin{array}{lll} \mathsf{nameof}(\mathsf{string}) = \mathsf{String} & \mathsf{nameof}(\mathsf{number}) = \mathsf{Number} & \mathsf{nameof}(\mathsf{record}) = \mathsf{Record} \\ \mathsf{nameof}(\mathsf{bool}) = \mathsf{Boolean} & \mathsf{nameof}(\mathsf{collection}) = \mathsf{List} & \mathsf{nameof}(\mathsf{named}\ \nu) = \nu \end{array}$$

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

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

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

$$[\![\bot]\!]_e = [\![\mathsf{null}]\!]_e = \mathsf{Data}, e, \emptyset$$

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

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

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

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

we get corresponding F# types τ_i and generate n members of type $\mathsf{option}\langle\tau_i\rangle$. When the member is accessed, we need to perform a runtime type test using hasType to ensure that the value has the right type (similarly to runtime type conversions from the top type in languages like Java). If the type matches, a Some value is returned. The type inference algorithm also guarantees that there is only one case for each type tag (§3.3) and so we can use the tag when naming the generated member (using the nameof function).

Example 1. To illustrate how the mechanism works, we consider two examples. First, assume that the inferred type is a record { Age : $\mathsf{option}\langle\mathsf{int}\rangle$, Name : string }. The rules from Figure 6 produce the following class:

```
type Person(v : Data) =
  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 $\mathsf{getField}(\mathsf{Person}, \mathsf{Age}, v)$. If the returned field is not null, then the member calls asInt and wraps the result in Some. Note that getField is defined 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 $\mathsf{new}\ \mathsf{Person}(s)$.

Example 2. The second example illustrates the remaining types including collections and variants. Reusing Person from the previous example, consider $[\mathsf{any}\langle\mathsf{Person}, \mathsf{string}\rangle]$, a collection which contains a mix of Person and string values:

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

The type provider maps the structural type to an F# type $\mathsf{list}\langle\mathsf{PersonOrString}\rangle$. Given a structural document value s, the code to obtain the wrapped F# value is:

$$\mathsf{map}\ (\lambda x \to \mathsf{new}\ \mathsf{PersonOrString}(x))\ (\mathsf{getChildren}(s))$$

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 hasType. 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 6) 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 4 (Correctness of provided types)**.** *Given a sample value s and an input value s' such that $\langle\!\langle s \rangle\!\rangle \succ s'$ 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, analyse 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 hasType. □

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 [**?**], we prove type preservation (reduction does not change type) and progress (an expression that is not a value can be reduced).

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

For all new inputs s' such that $\exists i.(\langle\!\langle s_i \rangle\!\rangle \leq \langle\!\langle s' \rangle\!\rangle)$, 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 6) and progress (Lemma 7). □

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

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

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

The Lemma 4 guarantees that this does not happen inside the provided type. In Theorem 5, 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 7 (Progress)**.** *Given the assumptions and definitions from Theorem 5, 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 4. □

6. Practical experience

The F# Data library has been widely adopted by the industry 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 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 practicve

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 [**?**]. 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.

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

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[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 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 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 (3.4) 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):

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

We omit the details, but finding a preferred typing 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 [**? ? ? ?**], 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 [**? ?**] and databases [**? ?**], into a programming language, required the user to define the schema or it has an ad-hoc extension that reads the schema.

Cω [**?**] 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 [**?**] language has a rich system for working with records; meta-programming [**?**], [**?**] and multi-stage programming [**?**] could be used to generate code for the provided types; and gradual typing [**? ?**] 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 [**?**] 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 5) difficult. Extending our *relativized safety* to *probabilistic safety* is an interesting future work.

The PADS project [**? ?**] tackles a more general problem of handling *any* data format. The schema definitions in

[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.

PADS are similar to our structural type. The structure inference for LearnPADS [**?**] 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 preferred typing 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.