Making structured data first-class citizens

Tomas Petricek
University of Cambridge
tomas.petricek@cam.ac.uk

Don Syme
Microsoft Research, Cambridge
don.syme@microsoft.com

Abstract

Accessing data in structured formats such as XML, CSV and JSON in statically typed languages is difficult, because the languages do not understand the structure of the data. Dynamically typed languages make this syntactically easier, but lead to error-prone code. Despite numerous efforts, most of the data available on the web do not come with a schema. The only information available to developers is a set of examples, such as typical server responses.

We describe an inference algorithm that infers a type of structured formats including CSV, XML and JSON. The algorithm is based on finding a common supertype of types representing individual samples (or values in collections). We use the algorithm as a basis for an F# type provider that integrates the inference into the F# type system. As a result, users can access CSV, XML and JSON data in a statically-typed fashion just by specifying a representative sample document.

1. Introduction

We are witnessing an explosion of digital data. International organizations [5] and governments [2, 6] expose numerous datasets; knowledge is collected and schematized by communities [3] and numerous commercial services provide web services for data access[1].

Despite this fact, few strongly-typed programming languages are able to seamlessly integrate external information sources as if they were strongly-typed components from a programmer's perspective. The *type provider* mechanism of F# 3.0 [4] makes it possible to integrate external data sources as statically typed components through a compiler extensibility mechanism.

[1] As of the time of writing the `http://www.programmableweb.com` directory lists 8357 web APIs in over 60 categories.

Unpublished draft.

In this paper, we focus on accessing structured data formats such as CSV, XML and JSON. These formats are used by most of the aforementioned services. However, only a few of the services provide a schema or other specification of the formally defines structure they use. Most often programmers depend on unreliable documentation and exploration of sample responses.

In this paper, we combine type inference from sample documents with F# type provider mechanism. The two key contributions of this paper are:

- We introduce F# type providers for three most common structured document formats (CSV, XML and JSON) that make structured data first-class citizens in F# and make them accessible in a strongly-typed way.
- We define types for structured data formats and present a type inference algorithm based on the subtyping relation that infers type (or schema) of any structured data format from a sample or a collection of samples.

We do not expect familiarity with F# type providers, so the mechanism is introduced along with our structural type providers in Section 2. Type inference is presented in Section 3. Section 4 discusses runtime representation of structured values and proves that it is sound with respect to our static types. Finally, Section 5 connects the static and dynamic aspects by discussing how individual type providers (CSV, XML and JSON) work.

The work described in this paper is available as part of the F# Data library, which is an open-source project used in practice by a number of commercial F# users[2]. For this reason, we follow a pragmatic approach and do not ignore constraints of the real-world (such as the complexity of the host platform or the presence of null values).

2. Structural type providers

We start with a motivating example that shows how the F# Data library simplifies working with the JSON format and then introduce other structured type providers. At the same time, we introduce the F# 3.0 type provider mechanism and highlight the key features of the type inference.

[2] Available at `http://tpetricek.github.com/FSharp.Data`

2.1 Working with JSON documents

JSON is a popular format for data exchange on the web. It is based on data structures used in JavaScript and uses six types (Number, String, Boolean, Array, Object and Null). Consider the following example:

```
[ { "name" : null, "age" : 23},
  { "name" : "Alexander", "age" : 1.5},
  { "name" : "Tomas"} ]
```

In a statically typed (functional) language, JSON values would be represented using a single type. We could use pattern matching to check that the value matches an expected structure and then extract the required values. Assuming data is the above document, we print the names as follows:

```
match data with
| Array items →
  for item in items do
    match item with
    | Object keys →
      printf "%s" (Map.find keys "name")
    | _ → failwith "Incorrect format"
| _ → failwith "Incorrect format"
```

The code expects that the input is in a certain format (array of objects with the "name" field) and it fails if the input does not match these requirements. The code is complicated for two reasons. First, we have to repeatedly pattern match and explicitly handle failures and second, fields are accessed indirectly (using the name as a string rather than member of a statically known type). The first difficulty could be avoided by defining helper functions, but there is no way to avoid the second problem without mechanism such as type providers.

Assuming the `people.json` file contains the above data and data is a string containing information in the same format, we can rewrite the code using JsonProvider as follows (also printing the age, if it is present):

```
type People = JsonProvider<"people.json">

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

This code achieves the same simplicity as dynamically typed programming languages, but it is statically type-checked.

Type providers overivew. The parameter "people.json" is resolved statically at compile-time (it has to be a constant) and is passed to the JsonProvider which builds a specification of the People type and passes it to the F# compiler. This type information is also available at development time allowing advanced tooling such as code completion.

The JsonProvider uses a type inference algorithm (Section 3) to infer the JSON schema from the sample file and generates F# types that can be used to read the data. In this case, the resulting type is a collection of records with Name of type string and Age of type decimal option.

The type provider also specifies code that should be executed at run-time in place of item.Name and other operations (demonstrated in the next section). In this example, the runtime behaviour is the same as in the version using pattern matching, meaning that the member access throws an exception if the document does not have the expected format. We specify this precisely in Section 4.

The role of records. The example suggests that records play a prominent role when working with structured data. In fact, all three formats that we consider naturally contain a notion of record and we map them to F# object types with members to support code completion.

The provider infers the types of fields (string and decimal) and marks the Age field as optional. This is not the only alternative – we could infer a union (sum) type consisting of two different records (with one and two fields), but that would complicate the processing code. In general, our inference algorithm tries to minimize the number of unions in the inferred type.

2.2 Reading CSV files

The CSV format has the simplest structure of the formats that we consider. The structure is a collection of records with fields (with names specified by the first row). When inferring the type, we need to infer the type of fields. Consider the following example:

```
Ozone; Temp; Date
41;    67;   2012-05-01
36.3;  72;   2012-05-02
12.1;  74;   3 May
```

Assuming the `airdata.csv` file is available locally, we can use a type provider that prints data obtained from a web site with live CSV data as follows:

```
type AirCsv = CsvProvider<"airdata.csv">

let air = AirCsv.Load("http://data...air.csv")
for row in air do
  printf "%s: %d" row.Date row.Ozone
```

The type of the record (row) is inferred from the sample file, which contains three fields. Ozone is always a numeric value, so the provider infers a decimal type; Date uses mixed formats, so it is inferred as string.

Erasing type providers. At runtime, the type providers we describe use an erasure mechanism similar to Java Generics [1]. When called by the compiler, a type provider also returns code that is executed in place of the generated types. In the above example, the compiled code looks as follows:

```
let air = CsvFile.Load("airdata.csv")
for row in air.GetRows() do
  printf "%s %f" row.GetString("Date")
                 row.GetDecimal("Ozone")
```

The generated type AirCsv is erased to an underlying type CsvFile that represents any CSV document. Access to a named property, such as Ozone, is compiled into a method call, which reads a column using its name and converts it to a desired type (GetDecimal). Technical report [4] provides more details on type erasure.

2.3 Parsing RSS feeds

XML and JSON formats often contain collections and, especially in case of XML, the collections can be heterogeneous containing elements of different type (i.e. differently named XML nodes). Consider the following RSS feed:

```
<rss version="2.0"><channel>
  <title> BBC News - Europe </title>
  <item><title> Kurdish activists
      killed in Paris </title></item>
  <item><title> German MPs warn
      over UK EU exit </title></item>
</channel></rss>
```

The channel node contains a single title node and multiple item nodes. Moreover, title nodes always contain only a single child, which is a plain text.

When extracting data from similar XML documents, we often need to get nodes of a specific name or get the content of a node. The type provider we present is optimized for this style of access and allows processing the feed as follows:

```
type RssFeed = XmlProvider<"rss.xml">

let rss = RssFeed.Load("http://bbc.co.uk/...")
printf "%s " rss.Title
for item in rss.GetItems() do
  printf "%s " item.Title
```

In this example a heterogeneous collection (rss) is represented as a type that contains a member (Title) for an element that is present exactly once and a method (GetItems) for elements that are present multiple times. As a further simplification, if an element contains only a primitive value (such as Title) it is represented as a value of primitive type.

Collections and union types. As an alternative, the inference could produce a collection of sum type (with a case for item and a case for title). This alternative is useful for documents with complicated structure (such as XHTML files), but less useful for files with simpler structure such as web server responses, data stores and configuration files.

Moreover, our approach has a nice property that the following two, semantically equivalent, XML nodes are exposed a type with the same public interface:

```
<author name="Tomas" age="27" />
<author><name>Tomas</name><age>27</age></author>
```

In both cases, the type is a record with two properties, typed string and int respectively. The discussion in Section 3 uses the simple model (using sum types) and the inference described in the above examples is discussed in Section 6.1.

3. Structural type inference

The type inference algorithm for structured data is based on a subtyping relation. When inferring the type of a specified document, we infer (the most specific) types of individual values, such as CSV rows or JSON nodes, and then find the common supertype of values in a given dataset.

We define *structural type* τ which is a type of the structured data. Note that this type is distinct from the F# types (type provider maps *structural types* to ordinary F# types). The subtyping relation is not mapped to a subtyping relation between F# types, but it holds at runtime (meaning that an operation on a super-type is allowed on all subtypes). The operational aspects are discussed in Section 4.

3.1 Structural types

The grammar below defines *structural type* τ. Records and record fields are named with names ν. Record fields are marked with a qualifier δ which is either ? for optional fields or empty for required fields.

$$
\begin{aligned}
\tau \ ::= \ & \top \ \mid \ \textbf{null} \\
\mid \ & \tau + \ldots + \tau \ \mid \ [\tau] \\
\mid \ & \nu_{\mathsf{opt}} \ \{\delta_1\nu_1 : \tau_1, \ \ldots \ , \delta_n\nu_n : \tau_n\} \\
\mid \ & \mathsf{int} \ \mid \ \mathsf{decimal} \ \mid \ \mathsf{float} \ \mid \ \mathsf{bool} \ \mid \ \mathsf{string}
\end{aligned}
$$

In structured documents such as CSV, XML and JSON, the record type is the most prominent. We use it to type rows of a CSV file, the type of XML nodes and JSON objects.

To support these scenarios, records may be optionally named with ν_{opt}. For example, records representing XML nodes are named with the name of the node. Fields of a record are named with names $\nu_1, \ldots, \nu_n$ and each field may be marked as optional (we write ?foo for an optional field and foo for a mandatory field).

We start by using a simple collection type $[\tau]$ together with a union type $\tau + \ldots + \tau$. As discussed earlier, the library implements a more sophisticated type inference for heterogeneous collections, which is discussed in Section 6.1. Aside from collections, a union type appears when the user provides multiple samples with different structure.

Alternative representations. The record type in our definition is more complex than in other systems, so it is worth discussing the alternatives and justifying our design.

Firstly, we could use an optional type to represent op- Ref. some
tional fields. However, none of the structured formats we "other sys-
consider (CSV, XML, JSON) uses optional types elsewhere, tem" here?
so we could represent types that are not inhabited in any our use case. Secondly, we could replace named fields and (optionally) named records with a type assigning a name to any other type (such as τ **as** ν). Again, our use cases only name record types and their fields, so types such as [int] **as** ν would not be inhabited.

Finally, one of the crucial operations in the system is finding common types for multiple record values. For example,

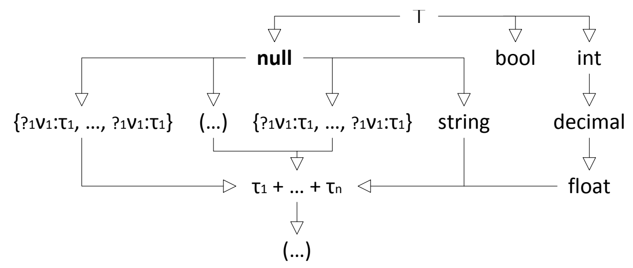

Figure 1. Subtype relation between structural types

for values $\{\mathsf{a} = 5\}$ and $\{\mathsf{a} = 10, \mathsf{b} = \textbf{true}\}$ we want to obtain a type $\{\mathsf{a} : \mathsf{int}, ?\mathsf{b} : \mathsf{bool}\}$. This would require complex rules matching records of named optional types (and records of optional named types).

3.2 Subtyping relation

To provide a basic intuition, the subtyping relation between structured types is informally demonstrated in Figure 1. The formal definition looks as follows:

Definition 1. *A subtyping relation between structural types (we write $\tau_1 :> \tau_2$ to denote that τ_2 is a subtype of τ_1) is defined as (a transitive reflexive closure of):*

$$\mathsf{float} :> \mathsf{decimal} :> \mathsf{int} \tag{1}$$

$$\tau :> \textbf{null} \quad (\text{iff } \tau \notin \{\top, \mathsf{bool}, \mathsf{int}, \mathsf{decimal}, \mathsf{float}\}) \tag{2}$$

$$\tau :> \top \quad (\text{for all } \tau) \tag{3}$$

$$[\tau_1] :> [\tau_2] \quad (\text{if } \tau_1 :> \tau_2) \tag{4}$$

$$\tau_1 + \ldots + \tau_n :> \tau \quad (\text{if } \exists i.\tau_i :> \tau) \tag{5}$$

$$\begin{gathered}\tau_1 + \ldots + \tau_n :> \tau'_1 + \ldots + \tau'_m \\ (\text{if } \forall i \in 1..m.\ \exists j \in 1..n \text{ such that } \tau_j :> \tau'_i)\end{gathered} \tag{6}$$

$$\begin{aligned}&\nu_{\mathsf{opt}}\ \{\delta_1\nu_1 : \tau_1,\ \ldots\ , \delta_n\nu_n : \tau_n\} :> \\ &\nu'_{\mathsf{opt}}\ \{\delta'_1\nu'_1 : \tau'_1,\ \ldots\ , \delta'_m\nu'_m : \tau'_m \\ &\quad (\text{if } \exists \text{ injective } p : \{1..m\} \to \{1..n\} \text{ such that} \\ &\quad \forall i \in 1..n.(i \notin p[\{1..m\}] \implies \delta_i = ?) \text{ and} \\ &\quad \forall i \in 1..m.(\tau_{p(i)} :> \tau'_i \wedge \delta'_i = ? \implies \delta_{p(i)} = ?))\end{aligned} \tag{7}$$

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

- Numeric types correspond to F# types. We aim to infer a single most precise numeric type that can represent all values from a sample dataset. We order types by the ranges they represent (1); int is 32-bit integer, decimal has higher precision than float, but smaller range.
- The **null** value is a valid value of all record and union types as well as string, but it is not a valid value of other primitive types, also called *value types* (2).
- There is a top type (3), but no bottom type. However it is possible to find common supertype of any two types, because a union type $\tau_1 + \ldots + \tau_n$ is a supertype of all its components (5).
- However, given two types, there may be multiple common supertypes that are not related by subtyping. For example, arising from rules (5) and (7).
- A record type is a supertype of another record type if it contains all its fields (optional fields have to remain optional) and all other fields are optional. Records are only related if they are both anonymous or if they have the same name (7).

Some of the aspects (such as **null** type and primitive numeric types) may appear overly complicated and F# specific. This paper aims to describe a realistic system rather than an (overly) simplified model.

Analogous problems will appear with most real-world languages (JVM languages have both **null** and multiple numeric types; OCaml has multiple numeric types and would have to represent **null** with an explicit option type).

3.3 Common supertype relation

As mentioned earlier, the structured type inference relies on a finding common supertype. However, the partially ordered set of types does not have a unique greatest lower bound. For example, record types $\{\mathsf{a} : \mathsf{int}\}$ and $\{\mathsf{b} : \mathsf{bool}\}$ have common supertypes $\{?\mathsf{a} : \mathsf{int}, ?\mathsf{b} : \mathsf{bool}\}$ and $\{\mathsf{a} : \mathsf{int}\} + \{\mathsf{b} : \mathsf{bool}\}$ which are unrelated. Our inference algorithm aims to infer records when possible (the first case). To do that, we use a *common supertype* relation:

Definition 2. *A* common supertype *of types τ_1 and τ_2 is a type τ (written $\tau_1 \nabla \tau_2 \vdash \tau$) obtained according to the inference rules in Figure 2.*

Notable aspects of the definition including the tag function are discussed below. The common supertype relation finds a single supertype (among several possibilities). The following theorem states that the found type is, indeed, a common supertype:

Theorem 1. *If $\tau_1 \nabla \tau_2 \vdash \tau$ then $\tau :> \tau_1$ and $\tau :> \tau_2$.*

Proof. By analysis of the inference rules in Figure 2. □

When finding a common supertype of two records (record), we return a record type that has the union of fields of the two arguments. The types of shared fields are common supertypes of their respective types. Fields that are present in only one record or are optional in one of the records are marked as optional in the result.

Finding a common supertype of union types (union-1) is more complicated. Our definition aims to add as few additional cases as possible, without creating nested union types (e.g. $\mathsf{int} + (\mathsf{bool} + \mathsf{decimal})$). This is done by grouping types that have a common supertype which is not a union type (in the previous example int with decimal and bool). We then find common supertype of each group and create union of these types (in our example $\mathsf{decimal} + \mathsf{bool}$).

Assume that the record fields are ordered to make k the largest index such that it holds that $\nu_i = \nu'_i \Leftrightarrow i \le k$.

$$\text{(record)}\ \frac{\tau_i \triangledown \tau'_i \vdash \tau''_i \quad (\forall i \in 1..k) \qquad \text{let } \delta''_i = ? \text{ iff } \delta_i = ? \vee \delta'_i = ?}{\begin{array}{c}\nu_{\mathsf{opt}}\ \{\delta_1\nu_1 : \tau_1,\ \ldots\ , \delta_n\nu_n : \tau_n\} \triangledown \nu_{\mathsf{opt}}\ \{\delta'_1\nu'_1 : \tau'_1,\ \ldots\ , \delta'_m\nu'_m : \tau'_m\} \vdash \\ \nu_{\mathsf{opt}}\ \{\delta''_1\nu_1 : \tau''_1,\ \ldots\ , \delta''_n\nu_k : \tau''_k, ?\nu_{k+1} : \tau_{k+1}, \ldots, ?\nu_n : \tau_n, ?\nu'_{k+1} : \tau'_{k+1}, \ldots, ?\nu'_m : \tau'_m\}\end{array}}$$

Assume that the union cases are ordered to make k the largest index such that it holds that $\mathsf{tag}(\tau_i) = \mathsf{tag}(\tau'_i) \Leftrightarrow i \le k$ and assume that m', n' are indices such that it holds that $\tau_i = \textbf{null} \Leftrightarrow i > n'$ and $\tau'_i = \textbf{null} \Leftrightarrow i > m'$.

$$\text{(union-1a)}\ \frac{\tau_i \triangledown \tau'_i \vdash \tau''_i \quad (\forall i \in 1..k) \qquad \exists i.(\tau_i = \textbf{null} \vee \tau'_i = \textbf{null}) \qquad \nexists i.(\textbf{null} :> \tau_i \vee \textbf{null} :> \tau'_i)}{(\tau_1 + \ldots + \tau_n) \triangledown (\tau'_1 + \ldots + \tau'_m) \vdash (\tau''_1 + \ldots + \tau''_k + \tau_{k+1} + \ldots \tau_{n'} + \tau'_{k+1} + \ldots + \tau'_{m'} + \textbf{null})}$$

$$\text{(union-1b)}\ \frac{\tau_i \triangledown \tau'_i \vdash \tau''_i \quad (\forall i \in 1..k) \qquad \text{Premise of (union-1a) does not hold}}{(\tau_1 + \ldots + \tau_n) \triangledown (\tau'_1 + \ldots + \tau'_m) \vdash (\tau''_1 + \ldots + \tau''_k + \tau_{k+1} + \ldots \tau_{n'} + \tau'_{k+1} + \ldots + \tau'_{m'})}$$

The remaining cases do not have any assumptions:

$$\text{(union-2)}\ \frac{\exists i.\mathsf{tag}(\tau_i) = \mathsf{tag}(\tau) \qquad \tau \triangledown \tau_i \vdash \tau'_i}{\tau \triangledown (\tau_1 + \ldots + \tau_n) \vdash (\tau_1 + \ldots + \tau'_i + \ldots + \tau_n)} \qquad \frac{\nexists i.\mathsf{tag}(\tau_i) = \mathsf{tag}(\tau)}{\tau \triangledown (\tau_1 + \ldots + \tau_n) \vdash (\tau_1 + \ldots + \tau_n + \tau)}$$

$$\text{(list)}\ \frac{\tau \triangledown \tau' \vdash \tau''}{[\tau] \triangledown [\tau'] \vdash [\tau'']} \qquad \text{(num)}\ \frac{\tau :> \tau'}{\tau \triangledown \tau' \vdash \tau} \quad \frac{\tau :> \tau'}{\tau' \triangledown \tau \vdash \tau_1} \ (\tau, \tau' \in \{\mathsf{int}, \mathsf{decimal}, \mathsf{float}\})$$

$$\text{(top)}\ \top \triangledown \tau \vdash \tau \quad \tau \triangledown \top \vdash \tau \qquad \text{(equal)}\ \tau \triangledown \tau \vdash \tau \qquad \text{(null)}\ \frac{\tau :> \textbf{null}}{\tau \triangledown \textbf{null} \vdash \tau} \quad \frac{\tau :> \textbf{null}}{\textbf{null} \triangledown \tau \vdash \tau_1}$$

If no other inference rules applies, then the following rule is used:

$$\text{(union-3)}\ \tau_1 \triangledown \tau_2 \vdash \tau_1 + \tau_2$$

Figure 2. Inference judgements that define the common supertype relation

The types are grouped by a *tag* that determines the kind of type (number, record, etc.) and is defined as:

$$\begin{array}{rl}\overline{\tau} ::= & \mathsf{string} \mid \mathsf{bool} \mid \mathsf{number} \\ & \mid \mathsf{record} \mid \mathsf{named}\ \nu \mid \mathsf{union} \mid \mathsf{list}\end{array}$$

The tag of a type is obtained using a function tag. It holds that two types with the same tag (which are not unions) have a common supertype that is also not a union:

$$\begin{array}{ll}\mathsf{tag} : \tau \to \overline{\tau} & \\ \mathsf{tag}(\mathsf{string}) & = \mathsf{string} \\ \mathsf{tag}(\mathsf{bool}) & = \mathsf{bool} \\ \mathsf{tag}([\tau]) & = \mathsf{list} \\ \mathsf{tag}(\tau + \ldots + \tau) & = \mathsf{union} \\ \mathsf{tag}(\mathsf{int}) = \mathsf{tag}(\mathsf{decimal}) = \mathsf{tag}(\mathsf{float}) & = \mathsf{number} \\ \mathsf{tag}(\{?_1\nu_1 : \tau_1,\ \ldots\ , ?_n\nu_n : \tau_n\}) & = \mathsf{record} \\ \mathsf{tag}(\nu\ \{?_1\nu_1 : \tau_1,\ \ldots\ , ?_n\nu_n : \tau_n\}) & = \mathsf{named}\ \nu\end{array}$$

The function is undefined for the $\top$ and **null** types, but this is not a problem because these types are never used as arguments in Figure 2. Assuming that no primitive value has a type $\top$, the top type is always eliminated in the (top) rule and so it cannot appear as a member of union in any of the (union) rules.

The **null** type is handled explicitly. When combining two unions, we only include **null** type explicitly if it is present in one of the original unions and none of the other types permit **null** as a valid value (union-1a), otherwise **null** is excluded.

Minimising unions. We stated earlier that the common supertype relation minimises the use of union types (by preferring common numeric types or common record types when possible). This property can be stated and proved formally:

Theorem 2. *If $\tau :> \tau_1$ and $\tau :> \tau_2$ and τ is not a union type and $\tau_1 \triangledown \tau_2 \vdash \tau'$ then τ' is not a union type.*

Proof. The only rule that introduces an union type is (union-3). By examining possible structures of τ, we see that the rule can only be used if there is no other supertype of both τ_1 and τ_2. □

4. Runtime representation

In the previous section, we defined static type of structured documents and we defined the subtyping relation between types. In this section, we add the operational semantics – we describe runtime representation of values and discuss when a value belongs to a type. This also clarifies what structural changes in the input document do not break programs written using inferred types.

As discussed earlier, the structured type providers discussed in this article use the *type erasure* mechanism. At runtime, all values are represented using a single type that provides a number of operations that view the value as a value with a certain structure. If a value does not match the required structure, the operation is undefined.

```
let GetFloat value =
  match value.GetNumber() with
  | Choice1 n → floatOfInt n
  | Choice2 d → floatOfDecimal d
  | Choice3 f → f
```

```
let GetDecimal value =
  match value.GetNumber() with
  | Choice1 n → decimalOfInt n
  | Choice2 d → d
  | _ → ⊥
```

```
let GetInt value =
  match value.GetNumber() with
  | Choice1 n → n
  | _ → ⊥
```

Figure 3. Auxiliary functions for accessing numeric values

The key claim that we make in this (and the next) section is that, if we have an (inferred) type τ_1 and a value that does not contain **null** and belongs to a type τ_2 which is a subtype of τ_1 ($\tau_2 :> \tau_1$), then all operations that may be called by user code to access the value are defined.

At runtime, a value from a structured document is represented using a single type that we call Value[3]. The type is a record of functions that can be called to obtain a value with a specified format (and may be undefined if the required format is not available):

$$
\begin{array}{lcl}
\mathsf{GetBool} & : & \mathsf{Value} \rightarrow \mathsf{bool} \\
\mathsf{GetNumber} & : & \mathsf{Value} \rightarrow \mathsf{int} + \mathsf{decimal} + \mathsf{float} \\
\mathsf{GetString} & : & \mathsf{Value} \rightarrow \mathsf{string} \\
\mathsf{IsNull} & : & \mathsf{Value} \rightarrow \mathsf{bool} \\
\mathsf{GetItems} & : & \mathsf{Value} \rightarrow \mathsf{Value\ list} \\
\mathsf{GetField} & : & \mathsf{Value} \times \mathsf{string\ option} \times \mathsf{string} \\
 & & \quad \rightarrow \mathsf{Value\ option}
\end{array}
$$

The first three operations extract a primitive value. Note that GetNumber may return any of the supported numeric types. The F# runtime provides a conversion from int to decimal and from decimal to float. These are used in Figure 3 to define functions that get a value of a specific numeric type and convert values. The conversion may lose precision, but it does not overflow.

The GetField operation returns the value of a field – it takes string option which is the name of the desired record type (or None if the record is anonymous) and the name of the field. The operation returns None if the value is a record with matching name, but the field is missing.

[3] Here we slightly diverge from the actual representation in the F# Data library, which uses a different representation for every format, although they mostly share the structure discussed here.

Type of values. Next, we define what does it mean for a runtime Value to have a type τ and show that a value of a certain type is also a value of all its supertypes.

Definition 3. *Given a* Value ρ*, the value belongs to a type* τ *(written* $\rho \in \tau$*) as defined by inference rules in Figure 4 (we write* $f(x)\downarrow$ *to mean that* $f(x)$ *is defined).*

The definition matches the intuitive understanding presented so far. For numeric types, a value is of a specific type (e.g. decimal) if GetNumber returns the exact type or a type that can be converted to it (e.g. from int). When IsNull is true, the value belongs to any record and union type, as well as to the string type.

A value is of a union type if it belongs to any of the types that form the union. In order to belong to a specific record type, the GetField function needs to be defined for all fields and it has to return Some for all required fields. Finally, no value belongs to the $\top$ type.

Flexibility. When inferring document type from a sample, we can expect that future inputs will not have exactly the same structure. For example, web services may include additional data in the result.

- Using smaller number is allowed (but not bigger)
- Using null in place of string/record/union
- Using subtypes as elements of a collection

$$
\frac{\mathsf{GetInt}(\rho)\downarrow}{\rho \in \mathsf{int}} \quad
\frac{\mathsf{GetDecimal}(\rho)\downarrow}{\rho \in \mathsf{decimal}} \quad
\frac{\mathsf{GetFloat}(\rho)\downarrow}{\rho \in \mathsf{float}} \quad
\frac{\mathsf{GetBool}(\rho)\downarrow}{\rho \in \mathsf{bool}} \quad
\frac{\mathsf{IsNull}(\rho) = \mathbf{true}}{\rho \in \mathbf{null}}
$$

$$
\frac{\rho \in \mathbf{null} \vee \mathsf{GetString}(\rho)\downarrow}{\rho \in \mathsf{string}} \quad
\frac{\rho \in \mathbf{null} \vee (\mathsf{GetItems}(\rho)\downarrow \wedge \forall \rho' \in \mathsf{GetItems}(\rho).\rho' \in \tau)}{\rho \in [\tau]} \quad
\frac{\rho \in \mathbf{null} \ \vee \exists i.\rho \in \tau_i}{\rho \in \tau_1 + \ldots + \tau_n}
$$

$$
\frac{\rho \in \mathbf{null} \vee \forall i \in 1..n.\big((\mathsf{GetField}(\rho, \nu_{\mathsf{opt}}, \nu_i) = \mathsf{Some}(\rho') \ \wedge \ \rho' \in \tau_i) \vee (\mathsf{GetField}(\rho, \nu_{\mathsf{opt}}, \nu_i) = \mathsf{None} \ \wedge \ \delta_i \ = \ ?)\big)}{\rho \in \nu_{\mathsf{opt}} \ \{\delta_1\nu_1 : \tau_1, \ \ldots \ , \delta_n\nu_n : \tau_n\}}
$$

Figure 4. Inference rules that define type of a runtime value

- Adding fields is fine (or, in fact, having records that contain fields associated with multiple different record names)

Soundness of subtyping. The type inference algorithm proceeds by finding a common supertype of sample values. In order for this to be valid, a runtime value that has a type τ also needs to belong to all supertypes of τ. This is proved by the following theorem:

Theorem 3. *Assuming $\tau_1 :> \tau_2$ and a value ρ has a type τ_2 (written $\rho \in \tau_2$) then the value ρ has a type τ_1 ($\rho \in \tau_1$).*

Proof. By analysis of the cases in the subtyping relation as defined in Section 3.2. Assuming runtime requirements hold for τ_2, the runtime requirements for τ_1 hold as well. $\square$

5. Individual providers

now we have type inference and runtime representation, so we can connect the two (somehow..)

5.1 CSV files

Given a CSV file

Theorem 4. *Given a CSV file with inferred type τ, and a value ρ that belongs to this type, all accessors in the code written by the user are always defined.*

5.2 JSON documents

6. Additional stuff

6.1 Collections

6.2 Units of measure

References

[1] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. In *Proceedings of OOPSLA '98*, 1998.

[2] Data.gov.uk. United Kingdom Government. `http://data.gov.uk/`, 2013.

[3] Google. Freebase. `http://www.freebase.com/`, 2013.

[4] 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. F# 3.0 - strongly-typed language support for internet-scale information sources. Technical report, Microsoft Research, 2012.

[5] The World Bank. Data. `http://data.worldbank.org/`, 2013.

[6] United States Government. Data.org. `http://data.gov/`, 2013.