F# Data: Making structured data first-class citizens

Tomas Petricek
University of Cambridge
tomas@tomasp.net

Abstract

Most statically typed languages assume that programs run in a closed world, but this is not the case. Modern applications interact with external services and often access data in structured formats such as XML, CSV and JSON. Static type systems do not understand such external data sources and only make data access more cumbersome. Should we just give up and leave the messy world of external data to dynamic typing and runtime checks?

Of course, we should not give up! In this paper, we show how to integrate external data sources into the F# type system. As most real-world data on the web do not come with an explicit schema, we develop a type inference algorithm that infers a type from representative samples. Next, we use the type provider mechanism for integrating the inferred structures into the F# type system.

The resulting 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 incorrect closed world assumption.

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

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

1. Introduction

The key function of many modern applications is to connect multiple data sources or services and present the aggregated data in some form. Mobile applications for taking notes, searching train schedules or finding tomorrow's weather all communicate with one or more services over the internet.

Increasing number of such services provide REST-based endpoints that return data as CSV, XML or JSON. Despite numerous schematization efforts, the services typically do not come with schema. At best, the documentation provides a number of typical requests and sample responses.

For example, the OpenWeatherMap service provides an endpoint for getting current weather for a given city[1]. The page documents the input URL parameters and shows one sample JSON response to illustrate the response structure. Using a standard library for working with JSON and HTTP, we might call the service and read the temperature as follows [2]:

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

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

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

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

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

In the rest of the paper, we give more detailed description of the mechanism and we also discuss theoretical safety properties of the approach. The key novel contributions of this paper are:

- We present type providers for XML, CSV and JSON (Section 2) that are available in the F# Data library and we also cover practical aspects of the implementation that contributed to their industrial adoption (Section 5).
- We describe a predictable type inference algorithm for structured data formats that underlies the type providers (Section 3). It is based on finding common supertype of a set of examples.
- We introduce the notion of relativized type safety and show how it holds for the code written using our type providers (Section 4). Although we focus on our specific case, this demonstrates a new way of thinking about ML-style type safety that is needed in the context of web.

Although the F# Data library has been widely adopted is one of the most downloaded F# libraries it has not yet been presented in an academic form. Additional documentation and source code can be found at http://fsharp.github.io/FSharp.Data.

[1] See "Current weather data": http://openweathermap.org/current
[2] We abbreviate the full URL: http://api.openweathermap.org/data/2.5/weather?q=Prague&units=metric

2. Structural type providers

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

- Our mechanism is robust and predictable. This is important as the user directly works with the inferred types and should understand why a specific type was inferred from a given sample[3].
- Our inference mechanism prefers records over unions. This better supports developer tooling – most F# editors provide code completion hints on "." and so types with properties (records) are easier to use than types that require pattern matching.
- Finally, we handle a number of practical concerns that may appear overly complicated, but are important in the real world. This includes support for different numerical types, null values and missing data and also different ways of representing Booleans in CSV files.

2.1 Working with JSON documents

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

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

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

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

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

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

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

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

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

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

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

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

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

The type Entity represents the person. The property Name is optional, because one of the sample records contains null as the name. The property Age is also optional, because the value is missing in one sample. The two sample ages are an integer 25 and a decimal 3.5 and so the common inferred type is decimal.

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

Error handling. In addition to the structure of the types, the type provider also specifies what code should be executed at run-time in place of item.Name and other operations. This is done through a *type erasure* mechanism demonstrated in Section 2.2. In this example, the runtime behaviour is the same as in the hand-written sample in Section 1 – a member access throws an exception if the document does not have the expected format.

Informally, the safety property discussed in Section 4 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 – for example, if the OpenWeatherMap changes the response format, the sample in Section 1 will not re-compile and the user knows that the code needs to be changed. What this means for the traditional ML type safety is discussed in Section 4.1.

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

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

2.2 Reading CSV files

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

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

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

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

```
type AirCsv = CsvProvider⟨"airdata.csv"⟩

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

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

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

Erasing type providers. At runtime, the type providers we describe use an erasure mechanism similar to Java Generics [?]. A type provider also generates code that is executed in place of the members of the generated types. In the above example, the compiled (and actually executed) code looks as follows:

```
let air = CsvFile.Load("airdata.csv", fun r →
  asDecimal r.[0], asIntOpt r.[1], r.[2], asBool r.[3])

for c1, _, c3, c4 in air.Rows do
  if not c4 then printf "%s: %d" c3 c1
```

The generated type AirCsv is erased to a type CsvFile⟨α⟩. The Load method takes the file name together with a function that turns a row represented as an array of strings to a typed representation – here, a four-element tuple decimal ∗ option⟨int⟩ ∗ string ∗ bool (this type is also used as a type argument of CsvFile⟨α⟩. The properties such as Ozone are then replaced with an accessor that reads the corresponding tuple element.

More details about the full type erasure mechanism can be found in the paper on F# type providers [?]. We present a simplified model when discussing safety in Section 4.

2.3 Processing XML data

The XML format is similar to JSON in that it uses nested elements rather than a flat CSV-like structure. In our inference algorithm, we use the notion of *records* for both JSON records and XML elements. However, unlike in JSON, the XML elements also have a name. An element can contain a collection of child elements. In case of XML, the collection can often be heterogeneous containing child nodes of different names. Consider the following (simplified) RSS news feed as an example:

```
<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 (with the name of the feed) and multiple item nodes (individual news articles). In RSS, the title nodes always contain plain text, while item nodes contain title (and also url and othe properties omitted here).

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

```
type RssFeed = XmlProvider⟨"rss-sample.xml"⟩

let channel = RssFeed.Load("http://bbc.co.uk/...").Channel
printf "News from: %s " channel.Title
for item in channel.Items do
  printf " - %s " item.Title
```

In this example, the type of the channel value represents a heterogeneous collection that contains exactly one title node and zero or more item nodes.

Heterogeneous collections and unions. Our inference algorithms recognizes this as a *heterogeneous collection* (if the same structure appears in all samples) and generates a type with a member Title for accessing the value of the singleton title element together with a member Items that returns a collection of zero or more items. As a further simplification, if an element contains only a primitive value (such as title) it is represented as a value of primitive type (so channel.Title has a type string).

As an alternative, the inference could produce a collection of union 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 based on heterogeneous collections has a nice property that it generates types with the same public interface for the following two ways of representing data in XML:

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

In both cases, the provided type for author has two properties, typed string and int respectively. In Section 3, we discuss the simplified model (using union types). The inference based on heterogeneous collections is described later in Section 5.

2.4 Summary

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

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

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

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

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

3. Structural type inference

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

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

Note that the subtyping relation between structural types σ does not map to a subtyping relation between F# types τ. However, it does hold at runtime, meaning that an operation on a supertype can be performed on its subtypes. We make the type provider mechanism and runtime aspects precise in Section 4 when discussing safety.

3.1 Structural types

The grammar below defines *structural type* σ. We distinguish between *non-nullale 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 ν_{opt} for record names, which can be empty:

$$
\begin{aligned}
\hat{\sigma} &= \nu_{\mathsf{opt}}\ \{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\} \\
&\quad \mid \mathsf{float} \mid \mathsf{decimal} \mid \mathsf{int} \mid \mathsf{bit} \mid \mathsf{bool} \mid \mathsf{string} \\
\sigma &= \hat{\sigma} \quad \mid \hat{\sigma}\ \mathsf{option} \\
&\quad \mid [\sigma] \mid \sigma_1 + \ldots + \sigma_n \\
&\quad \mid \top \quad \mid \mathsf{null}
\end{aligned}
$$

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

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

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

Next, we have three numerical types – `int` for integers, `decimal` for small high-precision decimal numbers and `float` for floating-point numbers (these are related by sub-typing relation as discussed in the next section). Furthermore, the `bit` type is a type of values 0 and 1 and makes it possible to infer Boolean in the CSV processing example discussed in Section 2.2.

Any non-nullable type is also a nullable type, but it can be wrapped in the `option` constructor to explicitly permit the `null` value. These are typically mapped to the standard F# option type. A simple collection type $[\sigma]$ is also nullable and missing values or `null` are treated as empty collection. The type `null` is inhabited by the `null` value (using an overloaded but not ambiguous notation) and $\top$ represents the top type.

Finally, a union type in our model implicitly permits the `null` value. This is because structural union types are *not* mapped to F# union types. Instead our encoding requires the user to handle the cases one-by-one and so they also need to handle the situation when none of the cases matches (as discussed in Section 5, this is the right choice for an F# type provider, but it would not necessarily be a good fit for other languages).

3.2 Subtyping relation

To provide a basic intuition, the subtyping relation between structured types is illustrated in Figure 1. We split the diagram in two parts. The upper part shows non-nullable types (with records and primitive types). The lower part shows nullable types with `null`, collections and optional values. We omit links between the two part, but any type $\hat{\sigma}$ is a subtype of $\hat{\sigma}$ `option` (in the diagram, we abbreviate σ `option` as σ?).

The diagram shows only the basic structure and it does not explain relationships between records with same name, but different fields and between union types. This following definition makes this precise.

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

Primitives, options, collections

$$\texttt{float} :> \texttt{decimal} :> \texttt{int} :> \texttt{bit} \quad \text{and} \quad \texttt{bool} :> \texttt{bit} \qquad \text{(B1)}$$
$$\sigma :> \texttt{null} \quad (\text{iff } \sigma \neq \hat{\sigma}) \qquad \text{(B2)}$$
$$\hat{\sigma}\ \texttt{option} :> \hat{\sigma} \quad (\text{for all } \hat{\sigma}) \qquad \text{(B3)}$$
$$\sigma :> \top \quad (\text{for all } \sigma) \qquad \text{(B4)}$$
$$[\sigma_1] :> [\sigma_2] \quad (\text{if } \sigma_1 :> \sigma_2) \qquad \text{(B5)}$$

Union types

$$\sigma_1 + \ldots + \sigma_n :> \sigma'_1 + \ldots + \sigma'_n \quad (\text{iff } \forall i.\sigma_i :> \sigma'_i) \qquad \text{(U1)}$$
$$\sigma_1 + \ldots + \sigma_n :> \sigma_1 + \ldots + \sigma_m \quad (\text{where } m \leq n) \qquad \text{(U2)}$$
$$\sigma_1 + \ldots + \sigma_n :> \sigma_{\pi(1)} + \ldots + \sigma_{\pi(n)} \quad (\text{for a permutation } \pi) \qquad \text{(U3)}$$
$$\sigma_1 + \ldots + \sigma_n :> \sigma_i \quad (\text{for } i \in \{1 \ldots n\}) \qquad \text{(U4)}$$

Record types

$$\nu\ \{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\} :> \nu\ \{\nu_1 : \sigma'_1, \ldots, \nu_n : \sigma'_n\} \quad (\sigma_i :> \sigma'_i) \qquad \text{(R1)}$$
$$\nu\ \{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\} :> \nu\ \{\nu_1 : \sigma_1, \ldots, \nu_m : \sigma_m\} \quad (m \geq n) \qquad \text{(R2)}$$
$$\nu\ \{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\} :> \nu\ \{\nu_{\pi(1)} : \sigma_{\pi(1)}, \ldots, \nu_{\pi(m)} : \sigma_{\pi(m)}\} \quad (\text{for a permutation } \pi) \qquad \text{(R3)}$$
$$\nu\ \{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n, \nu_{n+1} : \texttt{null}, \ldots, \nu_{n+m} : \texttt{null}\} :> \nu\ \{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\} \qquad \text{(R4)}$$

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

- For numeric types (B1), we infer a single most precise numeric type that can represent all values from a sample dataset, so we order types by the ranges they represent; `int` is a 32-bit integer, `decimal` has a range of about -1^{29} to 1^{29} and `float` is a double-precision floating-point.
- The `bit` type is a subtype of both `int` and `bool` (B1). Thus, a sample 0, 1 has a type `bit`, but 0, 1, `true` is `bool` and 0, 1, 2 becomes `int` (for a sample 0, 1, 2, `true` we would need a union type).
- The `null` type is a subtype of all nullable types (B2), that is all σ types excluding non-nullable types $\hat{\sigma}$). Any non-nullable type is also a subtype of its optional version (B3).

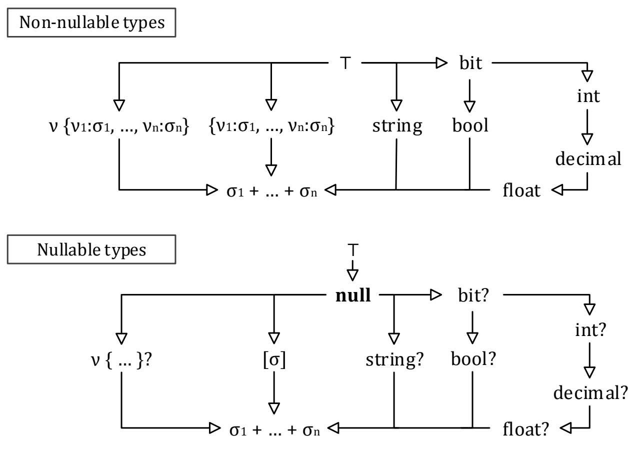

Figure 1. Subtype relation between structural types

- There is a top type (B4), 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 (U4).
- The remaining rules for unins are standard. Subtype can have fewer alternatives (U1), elements can be reordered (U2) and subtyping is covariant (U3). Similarly, the collection type is also covariant (B5).
- As usual, the subtyping on records is covariant (R1), subtype can have additional fields (R2) and fields can be reordered (R3). The interesting rule is the last one (R4). Together with covariance, it states that a subtype can omit some fields, provided that their types are nullable.

The rule that allows subtype to have fewer record elements (R4) is particularly important. It allows us to prefer records in some cases. For example, given two samples {`name` : `string`} and {`name` : `string`, `age` : `int`}, we can find a common supertype {`name` : `string`, `age` : `int option`} which is also a record. For usability reasons, we prefer this to another common supertype {`name` : `string`} + {`name` : `string`, `age` : `int`}. The next section describes precisely how our inference algorithm works.

It is worth noting that some of the aspects (such as 3 different numeric types and handling of the `null` type) are specific to F#. Similar problems will appear with most real-world languages and systems. JVM and OCaml all haves multiple numeric types, but they would handle `null` differently.

We could also be more or less strict about missing data – we choose to handle missing values silently when we can (`null` becomes an empty collection), but we are explicit in other cases (we infer `string option` as a hint to the user rather than treating `null` as an empty string). These choices are based on experience with using F# Data in practice.

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 {`a` : `int`} and {`b` : `bool`} have common supertypes {?`a` : `int`, ?`b` : `bool`} and {`a` : `int`}+{`b` : `bool`} which are unrelated. Our inference algorithm always prefers records over unions (Theorem 3).

The definition of the *common supertype* relation uses a number of auxiliary definitions that are explained in the discussion that follows.

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

When finding a common supertype of two records (*record-1*), we return a record type that has the union of fields of the two arguments. We assume that the names of the first *k* fields are the same and the remaining fields have different names. To get a record with the right order of elements, we provide the (*order*) rule. The types of shared fields become common supertypes of their respective types (recursively). Fields that are present in only one record are marked as optional using the following helper definition:

$$\lceil \hat{\sigma} \rceil = \hat{\sigma}\ \texttt{option} \qquad \text{(non-nullable types)}$$
$$\lceil \sigma \rceil = \sigma \qquad \text{(otherwise)}$$

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

$$\text{(union-1)}\ \frac{\exists i.\mathsf{tagof}(\sigma_i) = \mathsf{tagof}(\sigma) \quad \sigma \triangledown \sigma_i \vdash \sigma'_i \quad \mathsf{tagof}(\sigma) \neq \mathsf{union}}{\sigma \triangledown (\sigma_1 + \ldots + \sigma_n) \vdash (\sigma_1 + \ldots + \lfloor \sigma'_i \rfloor + \ldots + \sigma_n)} \qquad \frac{\nexists i.\mathsf{tagof}(\sigma_i) = \mathsf{tagof}(\sigma) \quad \mathsf{tagof}(\sigma) \neq \mathsf{union}}{\sigma \triangledown (\sigma_1 + \ldots + \sigma_n) \vdash (\sigma_1 + \ldots + \sigma_n + \lfloor \sigma \rfloor)}$$

$$\text{(union-2)}\ \frac{\mathsf{tagof}(\sigma_i) = \mathsf{tagof}(\sigma'_i) \quad \sigma_i \triangledown \sigma'_i \vdash \sigma''_i \qquad (\forall i \in \{1..k\})}{\begin{array}{l}(\sigma_1 + \ldots + \sigma_k + \ldots + \sigma_n) \triangledown (\sigma'_1 + \ldots + \sigma'_k + \ldots + \sigma'_m) \vdash \\ (\sigma''_1 + \ldots + \sigma''_k + \sigma_{k+1} + \ldots + \sigma_n + \sigma'_{k+1} + \ldots + \sigma'_m)\end{array}} \qquad \text{(union-3)}\ \frac{\text{(no other rule applies)}}{\sigma_1 \triangledown \sigma_2 \vdash \lfloor \sigma_1 \rfloor + \lfloor \sigma_2 \rfloor}$$

$$\text{(order)}\ \frac{\nu\ \{\nu_1 : \sigma_1, \ldots, \nu_n : \sigma_n\} \triangledown \sigma \vdash \sigma'}{\nu\ \{\nu_{\pi(1)} : \sigma_{\pi(1)}, \ldots, \nu_{\pi(n)} : \sigma_{\pi(n)}\} \triangledown \sigma \vdash \sigma'} \qquad \frac{\sigma_1 + \ldots + \sigma_n \triangledown \sigma \vdash \sigma'}{\sigma_{\pi(1)} + \ldots + \sigma_{\pi(n)} \triangledown \sigma \vdash \sigma'} \qquad (\pi \text{ permutation})$$

$$\text{(list)}\ \frac{\sigma_1 \triangledown \sigma_2 \vdash \sigma}{[\sigma_1] \triangledown [\sigma_2] \vdash [\sigma]} \qquad \text{(prim)}\ \frac{\sigma_1 :> \sigma_2}{\sigma_1 \triangledown \sigma_2 \vdash \sigma_1} \qquad \begin{array}{l}(\text{when } \sigma_1, \sigma_2 \in \{\mathsf{bit}, \mathsf{int}, \mathsf{decimal}, \mathsf{float}\} \\ \quad \text{or } \sigma_1, \sigma_2 \in \{\mathsf{bit}, \mathsf{bool}\})\end{array}$$

$$\text{(sym)}\ \frac{\sigma_1 \triangledown \sigma_2 \vdash \sigma}{\sigma_2 \triangledown \sigma_1 \vdash \sigma} \qquad \begin{array}{ll}\text{(refl)}\ \ \sigma \triangledown \sigma \vdash \sigma & \text{(null-1)}\ \ \sigma \triangledown \mathsf{null} \vdash \sigma \quad (\sigma :> \mathsf{null}) \\ \text{(top)}\ \ \top \triangledown \sigma \vdash \sigma & \text{(null-2)}\ \ \sigma \triangledown \mathsf{null} \vdash \sigma\ \mathsf{option} \quad (\sigma :\not> \mathsf{null})\end{array}$$

Figure 2. Inference judgements that define the common supertype relation

Finding a common supertype of when one type is a union is more complicated. Our definiion aims to use a flat structure (avoid nesting unions) and limit the number of cases. This is done by grouping types that have a common supertype which is not a union type. For example, rather than inferring $\mathsf{int} + (\mathsf{bool} + \mathsf{decimal})$, our algorithm will find the common supertype of int and $\mathsf{decimal}$ (which is $\mathsf{decimal}$) and it will produce just $\mathsf{decimal}+\mathsf{bool}$.

To identify which types have a common supertype which is not a union, we group the types by a *tag*, which is defined as:

$$\begin{array}{rl}\mathsf{tag} & = \mathsf{string} \mid \mathsf{bool} \mid \mathsf{number} \mid \mathsf{union} \mid \mathsf{collection} \\ & \mid \mathsf{rec\text{-}anon} \mid \mathsf{rec\text{-}named}\ \nu\end{array}$$

The tag of a type is obtained using a helper function $\mathsf{tagof}(-) : \sigma \rightarrow \mathsf{tag}$:

$$\begin{array}{rl}\mathsf{tagof}(\mathsf{string}) & = \mathsf{string} \\ \mathsf{tagof}(\mathsf{bool}) & = \mathsf{bool} \\ \mathsf{tagof}(\mathsf{decimal}) = \mathsf{tagof}(\mathsf{float}) & = \mathsf{number} \\ \mathsf{tagof}(\mathsf{int}) = \mathsf{tagof}(\mathsf{bit}) & = \mathsf{number} \\ \mathsf{tagof}([\sigma]) & = \mathsf{collection} \\ \mathsf{tagof}(\sigma + \ldots + \sigma) & = \mathsf{union} \\ \mathsf{tagof}(\{\nu_1 : \sigma_1,\ \ldots\ , \nu_n : \sigma_n\}) & = \mathsf{rec\text{-}anon} \\ \mathsf{tagof}(\nu\ \{\nu_1 : \sigma_1,\ \ldots\ , \nu_n : \sigma_n\}) & = \mathsf{rec\text{-}named}\ \nu \\ \mathsf{tagof}(\hat{\sigma}\ \mathsf{option}) & = \mathsf{tagof}(\hat{\sigma})\end{array}$$

The function is undefined for the $\top$ and null types, but this is not a problem because these types never need to be used as arguments in Figure 2. The $\top$ type is always eliminated using the (*top*) rule and the null type is eliminated using either (*null-1*) or (*null-2*).

The handling of unions is specified using three rules. When adding a non-union type to a union (*union-1*), the union may or may not contain a case with the same tag as the new type. If it does, the new type is combined with the existing one, otherwise a new case is added. When combining two unions (*union-2*), we group the cases that have a shared tags. Finally, the last rule (*union-3*) covers the case when we are combining two non-union types. As discussed earlier, union implicitly permits null values and so we use the following auxiliary function which makes nullable types non-nullable (when possible) to simplify the type:

$$\begin{array}{rll}\lfloor \hat{\sigma}\ \mathsf{option} \rfloor & = \hat{\sigma} & \text{(option)} \\ \lfloor \sigma \rfloor & = \sigma & \text{(otherwise)}\end{array}$$

The remaining rules are mostly straightforward. For collections, we find the common supertype of its elements (*list*); for compatible primitive types, we choose one of the two (*prim*) and a common supertype with null is either the type itself (*null-1*) or an option type (*null-2*).

Properties. The common supertype relation finds a single supertype (among multiple possible candidates). The following theorems specify that the found type is uniquely defined (up to reordering of fields in records and union cases) and is, indeed, a common supertype.

Theorem 1 (Uniqueness)**.** *If $\sigma_1 \triangledown \sigma_2 \vdash \sigma$ and $\sigma_1 \triangledown \sigma_2 \vdash \sigma'$ then it holds that $\sigma :> \sigma'$ and $\sigma' :> \sigma$.*

Proof. The assumptions and shapes of the types to be unified in the rules in Figure 2 are disjoint, with the exception of (*order*), (*sym*) and (*refl*). These rules always produce types that are subtypes of each other. This property is preserved by rules that use the common supertype relation recursively. □

Theorem 2 (Common supertype)**.** *If $\sigma_1 \triangledown \sigma_2 \vdash \sigma$ then it holds that $\sigma :> \sigma_1$ and $\sigma :> \sigma_2$.*

Proof. By induction over the common supertype derivation using $\vdash$. □

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 3. *Given σ_1, σ_2 if there is s σ such that $\sigma :> \sigma_1$ and $\sigma :> \sigma_2$ and σ is not a union type, then $\sigma_1 \triangledown \sigma_2 \vdash \sigma'$ and σ' is not a union type.*

Proof. The only rule that introduces an union type is (*union-3*), which is used only when no other inference rule can be applied. □

4. Relativized type safety

4.1 Discussion

alternative - check the whole file upfront based on the inferred type what if the sample is online??

5. Implementation

Unions become records with fields - so having null makes sense
Write API
JSON - infering numbers from values (not just from actual value type - world bank example)
Heterogeneous collections

6. Conclusions

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...