F# Data: Making structured data first class citizens

Tomas Petricek
University of Cambridge, United Kingdom
tomas.petricek@cl.cam.ac.uk

1. Introduction

Despite numerous schematization efforts, most data is still available in structured formats – governments release data as CSV files, web services communicate using JSON and most XML files used in practice do not carry a schema. The only information that is usually available to developers is a set of examples, such as typical server responses in the documentation or data files from previous years.

Accessing such formats in statically typed languages is difficult. Only a few languages provide a mechanism for importing schema from external data sources as types. Even then, the absence of explicit schema makes such process hard. As a result, many developers working with data prefer dynamically typed languages that simplify data access, but make code more error prone.

Is it possible to access such structured external data in a statically typed way? In this paper, we present F# Data – an open source library of F# type providers for accessing XML, CSV and JSON data based on sample document(s) provided by the user. The library implements type inference algorithm that finds common supertype of the provided examples. Such inferred type becomes available through the type provider mechanism and recovers certain program properties.

2. Structural type providers

In this section, we demonstrate structural type providers from the F# Data (http://fsharp.github.io/FSharp.Data) which has become standard library for data acquisition in F#. We look at an example of reading data in the JSON format. Type providers are introduced along the way, so no previous knowledge of type providers is needed.

JSON is based on data structures used in JavaScript and uses six types (Number, String, Boolean, Array, Object and Null). The following snippet is a valid JSON document:

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

In a statically typed functional language, JSON documents can be represented using algebraic data type. When processing documents, we use pattern matching to extract the values that we expect to be available in the document. Assuming the above format, we could print the names of all persons in the document as follows:

```
match data with
| Array items →
   for item in items do
     match item with
     | Object prop → print (Map.find prop "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.

Although the above code requires a fixed structure (array of records with a certain field), it is written using abstractions that have been designed to allow handling non-fixed structures (such as pattern matching). If the input does not match the expected structure, the code simply throws an exception.

Assuming that `people.json` contains the above data and `data` is a string containing information in the same format, we can rewrite the code using `JsonProvider` from F# Data 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 "%d") item.Age
```

This code achieves the simplicity of dynamically typed languages, but in a typed way. The parameter `"people.json"` is special and is resolved statically at compile-time (it has to be a constant). It is passed to `JsonProvider` which contains code (executed at compile-time) that builds a specification of the `People` type and passes it back to the F# compiler. This type information is also available at development time allowing advanced tooling such as code completion.

The type provider also specifies code that should be executed at run-time in place of `item.Name` and other operations. To accommodate external data sources with huge number of types (such as Freebase and WorldBank in F# Data), type providers allow building *erased* types – types that exist only at type-checking phase, but disappear during the compilation e.g. `item.Name` is compiled to `item.GetStringProperty("name")`.

3. Type system for structured data

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. Structural types τ are defined as follows:

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

The type can be one of primitive types (numeric, string, Boolean), null and top type (which are needed for the `null` value and when type is not known – e.g. empty list).

The last three cases define a type of collections, type of (unlabeled) unions and a record type consisting of optional name ν (used by named XML elements, but ignored by JSON where records are not labeled) together with a collection of fields. Each field has a name ν_i and a type τ_i and an annotation δ_i specifying whether the field is optional or always present. This is directly supported in our algorithm because structured documents often contain missing values.

Subtyping relation. We do not attempt to provide formal definition here, but the key aspects of subtyping relation are:

- Numeric types correspond to F# types and we always infer the most precise numeric type. At runtime, `int` can be converted to `decimal`, which can be converted to `float`.
- The `null` value is valid for all union types, record types and `string` types (following .NET type system), but not for primitive numeric types.
- Record type is a subtype of another record type if it contains all its fields together with additional optional fields. Common supertype of two records is a record with the fields unique to one or the other marked as optional.
- There is a top type, but no single bottom type. Given two types, we can always find common supertype, because union type is supertype of all its components.

Example. To demonstrate how the type inference works in practice, let us revisit the example from Section 2. The sample document is a list of records with the `name` field (which may be `null`) and optional `age` field that is either `int` or `decimal`. The type provider maps the inferred type to the following F# types:

```
type Entity1 = list<Entity2>
type Entity2 =
  abstract Name : string
  abstract Age  : option<decimal>
```

Records are mapped to a type with named properties (whose implementation attempts to get the field by name and converts the value to the required type). Optional fields are mapped to `option` types and can then be handled using pattern matching or using function such as `Option.iter` used earlier.

Properties. The type generated based on a sample document can be used to read other documents of similar structure. Accessing a field of such document may fail when the document does not correspond to the inferred structure (e.g. field is missing or contains value that is not convertible to the desired type).

Consequently, our system does not have a traditional type soundness property. A relativized form of type soundness holds – when the input is subtype of all the provided sample(s), then the program will not fail. In practice, this turns out to be sufficiently strong guarantee for practical programming with XML, CSV and JSON formats.

Conclusions

In this paper, we gave a brief overview to the structural type providers in the F# Data library that simplify working with CSV, CML and JSON documents. Such documents often lack schema, which makes it difficult to integrate them into statically typed languages. Our approach is to infer the schema from sample document(s). This gives us a *relativized type safety* property, which recovers as much typing as possible, given the dynamic nature of the problem.

Related work

The F# Data library combines two aspects that have been considered separately in previous work. The first is integrating external data sources into statically typed language. LINQ [4] uses code generation and Links [1] achieves that by being tight to a specific data source (database). We built on top of F# type providers which have been described previously. The report [3] also provides more detailed overview of related work in that area.

The second component is type inference for structured formats. A related system has been presented in [2] which is designed to work over large-scale sample data sets and uses more heuristics to produce succinct type. In contrast, our approach is simpler, but works well for smaller samples that are often available when calling REST-based services or working with XML and CSV data.

1. Links: web programming without tiers. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. In the proceedings of FMCO 2006, LNCS 4709.
2. Typing Massive JSON Datasets. Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, In the proceedings of XLDI 2012
3. F#3.0 - Strongly-Typed Language Support for Internet-Scale Information Sources. Don Syme et al., Technical Report, MSR-TR-2012-101, Microsoft Research, 2012
4. LINQ: reconciling object, relations and XML in the .NET framework. Erik Meijer, Brian Beckman and Gavin Bierman. In the proceedings of SIGMOD 2006.