F# Data: Making structured data first-class citizens

Tomas Petricek[1], supervisor: Alan Mycroft[1]

[1] University of Cambridge, 15 JJ Thomson Avenue, CB3 0FD, UK
tomas.petricek@cl.cam.ac.uk, alan.mycroft@cl.cam.ac.uk

1. Motivation

Structured data formats such as XML, JSON and CSV are ubiquitous in modern software. Most data in the real-world come without an explicit schema. The only information available to developers is often a set of examples in the documentation. For example, consider the following JSON:

```
[ {"name": "Tomas"}, {"name": "Alexander", "age": 1.5},
  {"name": null, "age": 23}, {"name": "Miroslav", "age": 63 }]
```

The example is a list of records containing a `name` field (which may be `null`) and an optional `age` field (which may be an integer or a floating point). In ML-like languages we can print the names 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"
```

Although the code expects document of a particular format (array of objects with the `name` field), we had to write it using general pattern matching, leading to longwinded syntax. F# Data implements type inference that finds a common supertype of the specified JSON example(s). Such inferred type becomes available through the type provider mechanism, making the code shorter and easier to write (using auto-complete support in modern editors).

2. Background

F# Data combines two aspects that have been considered separately until now. The first is integrating external data into statically typed language. LINQ (Meijer, 2006) uses code generation; Links (Cooper, 2006) and *Cω* (Bierman, 2005) use a specific data source (database). We use F# type providers (Syme, 2012) which provide an extension point in the F# type system.

The second component is type inference for structured formats. (Colzano, 2012) presents a system designed to work over large-scale data sets and uses more heuristics to produce succinct type. 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.

3. Approach

Our type inference algorithm is based on subtyping. We infer the most specific types of individual values (such as CSV rows or JSON nodes) in a document and then find the common supertype of values in a given dataset. Structural types τ are defined as follows:

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

The type can be one of primitive types, null and top type. The interesting last three cases define a type of collections, type of (unlabeled) unions and records consisting of a collection of fields v_i with types τ_i and an annotation δ_i specifying whether the field is optional or always present.

The following shows three type inference rules that are crucial for the example above. We write $\tau_1 \nabla \tau_2$ in the assumption to denote the two unified types and the resulting type τ in the conclusion:

$$\frac{\{\ \delta_1 v_1 : \tau_1, \ldots, \delta_n v_n : \tau_n\ \} \nabla \{\ \delta_1 v_1 : \tau'_1, \ldots, \delta_n v_n : \tau'_n\ \}}{\{\ \delta_1 v_1 : \tau_1 \nabla \tau'_1, \ldots, \delta_n v_n : \tau_n \nabla \tau'_n\ \}}\ (\textit{recd-nest}) \qquad \frac{\text{float} \nabla \text{int}}{\text{float}}\ (\textit{prim-int})$$

$$\frac{\{\ v_1 : \tau_1, \ldots, v_n : \tau_n\ \} \nabla \{\ v_1 : \tau_1, \ldots, v_n : \tau_n, \ldots, v_{n+m} : \tau_{n+m}\ \}}{\{\ v_1 : \tau_1, \ldots, v_n : \tau_n, ?\, v_{n+1} : \tau_{n+1}, \ldots, ?\, v_{n+m} : \tau_{n+m}\ \}}\ (\textit{recd-opt})$$

The (*recd-nest*) rule specifies that two record types with matching fields are unified by recursively unifying the types of the fields. The (*prim-int*) rule allows unifying different numerical types (e.g. the type of a field with values 1.5 and 23 will be float). Finally, in the (*recd-opt*) rule the fields that are present only in one of the unified records ($v_{n+1}, \ldots, v_{n+m}$) are marked as optional in the resulting record.

4. Results and contributions

The JSON type provider in F# Data can be parameterized by a sample file, such as "people.json" and it generates types inferred from the sample document. The motivating example can be rewritten as follows:

```
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
```

Our algorithm infers that the document is a collection (and so it can be iterated using `for`) and each item has properties `Name` (which is string) and `Age` (which is optional). The code has the same correctness properties as our initial version, but is shorter and easier to write. Thus, our main contributions are:

- We use the type provider mechanism for integrating structured data into a language.
- We develop a simple, yet powerful, type inference for structured data formats.
- The F# Data library works well in practice and has been adopted by the industry for accessing XML data, calling REST-based services and processing CSV data[1].

References

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. *Links: web programming without tiers.* FMCO 2006

D. Colazzo, G. Ghelli, C. Sartiani. *Typing Massive JSON Datasets.* XLDI 2012

D. Syme et al. *F# 3.0 - Strongly-Typed Language Support for Internet-Scale Information Sources.* Technical Report, MSR-TR-2012-101, Microsoft Research, 2012

E. Meijer, B. Beckman and G. Bierman. *LINQ: Reconciling object, relations and XML in the .NET Framework*. In proceedings of SIGMOD 2006.

G. Bierman, E. Meijer, and W. Schulte. *The essence of data access in Cω.* ECOOP 2005.

[1] The F# Data package had 13,000 downloads at the time of writing. See also: http://fsharp.github.io/FSharp.Data