F# Data: Accessing structured data made easy

Tomas Petricek
University of Cambridge
tomas@tomasp.net

1. Introduction

Programming may not be the new *literacy*, but it is finding its way into many areas of modern society. For example, making sense of large amounts of data that are increasingly made available through open government data initiatives[1] is almost impossible without some programming skills. In media, *data journalism* reflects this development. Data journalists still write articles and focus on stories, but programming is at the core of their work.

Improving support for data access in programming language can make understanding data simpler, more usable and reproducible:

- **Simpler.** Many programming languages treat data as foreign entities that have to be parsed or processed. Instead, data should be treated as first-class entities fully integrated with the rest of the programming language.
- **Usability.** Modern developer tools make coding easier with auto-complete and early error checking. Unfortunately, these typically rely on static types which, in turn, make programming with standard untyped data harder.
- **Reproducible.** Data journalists often use a wide range of tools (including Excel, scripts and other ad-hoc tools). This makes it hard to reproduce the analysis and detect errors when the input data changes.

The presented work reconcilles the *simplicity* of data access in dynamically-typed programming languages with the *usability* and *reproducibility* provided by statically-typed languages. More specifically, we develop F# type providers for accessing data in structured data formats such as CSV, XML and JSON, which are frequently used by open government data initiatives as well as other web-based data sources.

2. Motivation: Accessing structured data

Despite numerous schematization efforts, most data on the web is available without an explicit schema. At best, the documentation provides a number of typical requests and sample responses. For simplicity, we demonstrate the problem using the OpenWeatherMap service, which can be used to get the current weather for a given city[2]. The page documents the URL parameters and shows one sample JSON response to illustrate the response structure.

Statically-typed. In a statically typed functional language like F#, we could use a library for working with HTTP and parsing JSON to call the service and read the temperature. Here, the parsing library returns a value of a JsonValue data type and we use pattern matching to extract the value we need[3]:

```
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 pattern matching assumes that the response has a particular format (as 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 a numerical value. When the format is incorrect, the data access simply fails with an exception.

The code is complicated, because data is parsed into a fully general data structure that we then process. The code is not benefiting from the generality of the data structure – quite the opposite!

Dynamically-typed. Doing the same in JavaScript is shorter and simpler (not surprisingly, as JSON has been designed after a subset of JavaScript). Using jQuery to perform the request, we can write:

```
jQuery.ajax("http://weather.org/?q=Prague",
  function(data) {
    var obj = JSON.parse(data);
    write("Lovely ", obj.main.temp, " degrees!");
  });
```

Although the code is shorter, writing it is not *easier* than writing the original statically-typed version. Even though some JavaScript editors provide auto-completion, they will fail to help us here, because they have no knowledge of the shape of the object obj. So, the author will have to open the documentation and guess the available fields from the provided sample.

Type providers. This paper presents the F# Data library that implements *type providers* for accessing structured data formats such as XML, JSON and CSV. Using the JSON type provider, we can write code with the same functionality in three lines, but with full editor support including auto-complete on the obj object:

```
type W = JsonProvider⟨"http://weather.org/?q=Prague"⟩
let obj = W.GetSample()
printfn "Lovely %f degrees!" obj.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 from the sample and provides a type that has a statically known property main, returning an object with a property temp that provides the temperature as a number.

This gives us the best of both worlds – the *simplicity* of dynamic typing with the *usability*, *safety* and associated tooling common in statically-typed languages.

[1] In the US (http://data.gov) and in the UK (http://data.gov.uk).

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

[3] We abbreviate the full URL: http://api.openweathermap.org/data/2.5/weather?q=Prague&units=metric

3. Background: Type providers

This paper presents a collection of *type providers* for integrating structured data into the F# programming language. As outlined in the previous example, our key technical contribution is the algorithm that infers appropriate type from an example document and and the type providers that expose the type. In this section, we give a brief overview of the type provider mechanims and of related approaches to integrating data into programming languages.

3.1 How type providers work

Documents in the JSON format consists of several possible kinds of values. The OpenWeatherMap example in the introduction used only (nested) record and a numerical value. To demonstrate other aspects, we look at a more complex example that also invloves collections and strings:

```
[ { "name": "Jan", "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. Assuming people.json contains the above sample and data is a string value that contains another data set in the same format, we can use JsonProvider 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
```

In contrast to the earlier example, the example now uses a local file people.json as a *representative sample* for the type inference, but then processes data (available at run-time) 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, 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 discussed below 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 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.

Erasing type providers. At compile-time, F# type providers use an erasure mechanism similar to Java Generics [1]. A type provider generates types and code that should be executed when members are accessed. In compiled code, the types are erased and the program directly executes the generated code. In the above example, the compiled (and actually executed) code looks as follows:

```
let items = asArray(JsonValue.Parse(data))
for item in items do
  printf "%s " asString (getProp "name" item)
  Option.iter (printf "(%f)")
    (Option.map asFloat (tryGetProp "age" item))
```

The generated type Entity is erased to a type JsonValue, which represents any JSON value and is returned by the Parse method. The remaining properties are erased to calls to various operations of the type provider runtime such as asArray, getProp or asFloat that attempt to convert a JSON value into the required structure (and produce a run-time exception if this is not possible).

The (hidden) type erasure process turns the static provided types into code that we might write without type providers. In particular, checked member names become unchecked strings. A type provider cannot remove all possibilities for a failure – indeed, an exception still occurs if the input does not have the right format, but it simplifies writing code and removes most errors when a representative sample is provided.

3.2 Type systems and data integration

The F# Data library connects two lines of research that have been previously disconnected. The first is extending the type systems of programming languages and the second is inferring the structure of real-world data sources.

The type provider mechanism has been introduced in F# [17, 18] and used in areas such as semantic web [14]. The library presented in this paper is the most widely used library of type providers and it is also novel in that it shows the programming language theory behind a concrete type provider.

Extending the type systems. A number of systems integrate external data formats into a programming language. Those include XML [8, 16] and databases [4]. In both of these, the system either requires the user to explicitly define the schema (using the host language) or it has an ad-hoc extension that reads the schema (*e.g.* from a database). LINQ [10] is more general, but relies on code generation when importing the schema.

The work that is most similar to F# Data is the XML and SQL integration in Cω [11]. It extends C# with types capable of representing structured data formats, but it does not infer the types from samples and it modifies the C# language (rather than using a general purpose embedding mechanism).

Aside from type providers, a number of other advanced type system features could be used to tackle the problem discussed in this paper. The Ur [2] language has a rich system for working with records; meta-programming [15], [5] and multi-stage programming [19] could be used to generate code for the provided types. However, 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. The second line of research related to our work focuses on inferring structure of real-world data sets. A recent work on JSON [3] infers a succinct type using MapReduce to handle large number of samples. It fuses similar types based on a type similarity measure. This is more sophisticated than our technique, but it would make formally specifying the safety properties difficult.

The PADS project [6, 9] tackles a more general problem of handling *any* data format. The schema definitions in PADS are similar to our structural type. The structure inference for LearnPADS [7] infers the data format from a flat input stream. A PADS type provider could follow many of the patterns we explore in our work, but formally specifying the safety property would be again challenging.

4. Approach and uniqueness

The F# Data library relies on two key techniques. First, it implements a type inference algorithm that generates a suitable type from one or more sample documents. Second, it implements a type provider that turns this *structural type* into an ordinary F# type that is then used by the programmers. The mapping preserves an important property that we call *relativized type safety*. Informally, we prove that when the provided sample is *representative*, code written using the type provider will not fail.

Whether a sample is *representative* is formally described using a subtyping relation on structural types σ. The full definition of the relationship can be found in our recent report [12]. In this paper, we provide a brief overview and discuss one interesting case in detail.

4.1 Structural types

The grammar below defines a *structural type* σ. 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:

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

The non-nullable types include records (consisting of zero or more fields with their types) and primitive types (int for integers, float for floating-point numbers, strings and Booleans).

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 the type provided for unions requires the user to handle the situation when none of the case matches (and so developers always provide code-path that can be run when the value is missing).

4.2 Subtyping relation

The subtyping relation between structural 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 following excerpt specifies some of the relationships formally:

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

$$\mathsf{float} :> \mathsf{int} \tag{1}$$
$$\sigma :> \mathsf{null} \quad (\text{iff } \sigma \neq \hat{\sigma}) \tag{2}$$
$$\hat{\sigma}\ \mathsf{option} :> \hat{\sigma} \quad (\text{for all } \hat{\sigma}) \tag{3}$$
$$\{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} :> \{\nu_1 : \sigma'_1, .., \nu_n : \sigma'_n\} \quad (\sigma_i :> \sigma'_i) \tag{4}$$
$$\{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} :> \{\nu_1 : \sigma_1, .., \nu_m : \sigma_m\} \quad (m \geq n) \tag{5}$$
$$\{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} :> \{\nu_{\pi(1)} : \sigma_{\pi(1)}, .., \nu_{\pi(m)} : \sigma_{\pi(m)}\} \tag{6}$$
$$\{\nu_1 : \sigma_1, .., \nu_n : \sigma_n, \nu_{n+1} : \mathsf{null}, .., \nu_{n+m} : \mathsf{null}\} :> \{\nu_1 : \sigma_1, .., \nu_n : \sigma_n\} \tag{7}$$

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

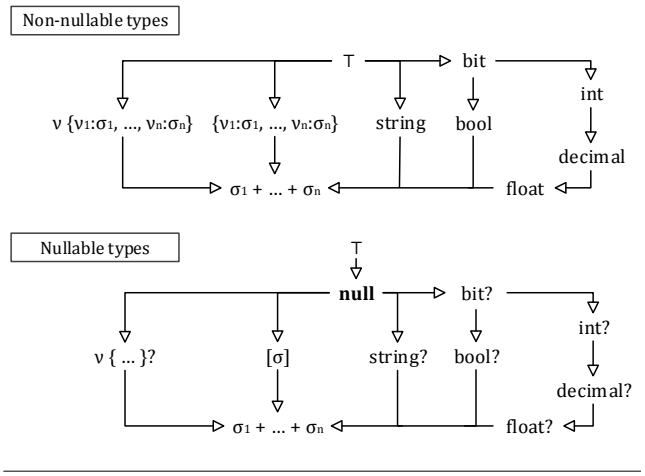

Figure 1. Subtype relation between structural types

- For numeric types (1), we infer the most precise numeric type that can represent all values from a sample dataset (even though this means loss of precision in some cases).
- The null type is a subtype of all nullable types (2), that is all σ types excluding non-nullable types $\hat{\sigma}$). Any non-nullable type is also a subtype of its optional version (2).
- The subtyping on records is covariant (4), subtype can have additional fields (5) and fields can be reordered (6). The interesting rule is (7) – 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 (8) is particularly important. It allows us to prefer records in some cases. For example, given two samples $\{\mathsf{name} : \mathsf{string}\}$ and $\{\mathsf{name} : \mathsf{string}, \mathsf{age} : \mathsf{int}\}$, we can find a common supertype $\{\mathsf{name} : \mathsf{string}, \mathsf{age} : \mathsf{int}\ \mathsf{option}\}$ which is also a record. For usability reasons, we prefer this to another common supertype $\{\mathsf{name} : \mathsf{string}\} + \{\mathsf{name} : \mathsf{string}, \mathsf{age} : \mathsf{int}\}$. Working with records does not require pattern matching and it makes it easier to explore data in editors that provide auto-completion.

4.3 Common supertype relation

As demonstrated by the example in the last section, structured types do not have a unique greatest lower bound. Our inference algorithm prefers records over unions and is defined in terms of the *common supertype* relation. The type infernce obtains a type for each sample (or each element of a collection) and then uses the relation to finds their common type. To demonstrate the idea, we show the definition for primitive types and records:

Definition 2. *A* common supertype *of types σ_1 and σ_2 is a type σ, written $\sigma_1 \triangledown \sigma_2 \vdash \sigma$, obtained according to the inference rules in Figure 2 (remaining rules can be found in the report [12]).*

When finding a common supertype of two records (*record*), 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 (other rules permit reordering of fields). 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:

$$\begin{aligned}\lceil\hat{\sigma}\rceil &= \hat{\sigma}\ \mathsf{option} && (\text{non-nullable types}) \\ \lceil\sigma\rceil &= \sigma && (\text{otherwise})\end{aligned}$$

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

$$\text{(list)}\ \frac{\sigma_1 \triangledown \sigma_2 \vdash \sigma}{[\sigma_1] \triangledown [\sigma_2] \vdash [\sigma]} \qquad \text{(prim)}\ \frac{}{\mathsf{int} \triangledown \mathsf{float} \vdash \mathsf{float}}$$

$$\text{(null-1)}\ \sigma \triangledown \mathsf{null} \vdash \sigma \quad (\sigma :> \mathsf{null})$$

$$\text{(null-2)}\ \sigma \triangledown \mathsf{null} \vdash \sigma\ \mathsf{option} \quad (\sigma :\not> \mathsf{null})$$

Figure 2. Selected inference judgements that define the common supertype relation

Discussing the full system is out of the scope of this paper, but the system has a number of desirable properties – the common supertype is uniquely defined and it is a supertype of both of the provided examples. Furthermore (to aid usability) our algorithm finds a common supertype that is not a union, if such type exists.

5. Results and contribution

The contributions and results of the presented work fall in three categories. First, our work is interesting in that it looks at a concrete type provider from the perspective of programming language theory and we introduce a novel notion of *relativized type safety*. Second, the F# Data library has become the most frequently downloaded F# library and is de-facto standard for data access in F#. Third, the work is also interesting from philosophical perspective because it illustrates an important change in thinking about types that we need to adopt when developing types for systems in the modern age of the web.

5.1 Relativized type safety

The safety property of F# Data 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, the user can always use the type provider with an input that has a different structure than any of the samples – and in this case, it is expected that the code will fail at runtime. Our formal model [12] states precisely what inputs can be handled by a provided type:

- Input can contain smaller numerical values (for example, if a sample contains float, the input can contain an integer).
- Records in the actual input can have additional fields.
- Records in the actual input can have fewer fields, provided that the type of the fields is marked as optional in the sample.
- Union types in the input can have both fewer or more cases.
- When we have a union type in the sample, the actual input can contain values of any of the union cases.

This is proved by our safety theorem, which relies on a simplified model of type providers and the F# language. We follow the approach of syntactic type safety [20] and show the type preservation (reducing expression does not change type) and progress (an expression that is not a value can be reduced).

Theorem 1 (Relativized safety)**.** *Assume $s_1, \ldots, s_n$ are samples, σ is a common supertype of the types of s_i. Next, assume that the type provider maps structural type σ to an F# type τ and that it provides a function f that takes an input s and produces a value τ.*

Then for all new inputs s' that are subtypes of one of the samples s_i, and for any expression e_c (user code) such that $x : \tau \vdash e_c : \tau'$, it is the case that $e_c[x \leftarrow f(s')] \rightarrow^* v$ for value v and $\vdash v : \tau'$.

5.2 New perspective on static typing

The *relativized safety* property does not guarantee the same amount of safety as standard type safety for programming languages without type providers. However, it reflects the reality of programming with external data sources that is increasingly important in the age of the web. In short, type providers do not *reduce* the safety – they simply *reveal* an existing issue.

In a separate paper [13], we argue that this is an important futrue direction for programming languages. Programs increasingly access external data sources (be it open government data accessed by data journalists or services consumed by mobile applications). Most programming language theory treats programs as closed expressions with no external dependencies – this gives us a formally tractable model, but it "throws out the baby with the bath water". Our experience with F# Data suggests that we can accept the reality of working with data *and* still provide clear formal model with provable safety properties.

5.3 Practical experience

The F# Data library has become the standard library for accessing data in F# and is widely used by both open-source projects and commercial users of F#. At the time of writing, it has 38 contributors and it has over 43,000 downloads[4].

The success of the library provides an evidence that some of the pragmatic (and perhaps controversial) choices made in the F# Data design work well in practice:

- The fact that a provided type may fail at run-time is not a problem as developers expect this when working with external data. In practice, the input that caused the issue can be added as another sample – the new compilation errors show which part of the program need to be modified to handle the input correctly.
- The F# Data library has been designed to work well with modern F# editors that use the F# compiler to provide autocompletion based on static types[5]. The success of F# Data confirms that editting experience and interactive development are important aspect of modern programming environments.

6. Conclusions

The F# Data library presented in this paper simplifies working with structured data formats such as XML, JSON and CSV. This is achieved by using the type provider mechanism to integrate external structured data into the programming language. We briefly outlined the programming language theory behind F# Data as well as its practical usability properties that contributed to its adoption.

Accessing and processing data is becoming an important ability in the modern information-rich society and we believe that tools such as F# Data can make programming simpler, more usable and more trustworthy – and, for example, enable data journalists produce analyses that their readers can not just understand, but also verify and modify.

[4] The first version (providing the functionality discussed here) has been created by the author of this paper; further contributions added new features and significantly improved the portability and overall quality of the library. See https://github.com/fsharp/FSharp.Data.

[5] This includes Visual Studio, Emacs, Xamarin Studio, Atom and others.

References

[1] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. *Acm sigplan notices*, 33(10):183–200, 1998.

[2] Adam Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *ACM Sigplan Notices*, volume 45, pages 122–133. ACM, 2010.

[3] Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. Typing massive json datasets. In *International Workshop on Cross-model Language Design and Implementation*, XLDI '12, 2012.

[4] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer, 2007.

[5] Jake Donham and Nicolas Pouillard. Camlp4 and Template Haskell. In *Commercial Users of Functional Programming*, 2010.

[6] Kathleen Fisher and Robert Gruber. PADS: a domain-specific language for processing ad hoc data. *ACM Sigplan Notices*, 40(6):295–304, 2005.

[7] Kathleen Fisher, David Walker, and Kenny Q. Zhu. LearnPADS: Automatic tool generation from ad hoc data. In *Proceedings of International Conference on Management of Data*, SIGMOD '08, pages 1299–1302, 2008.

[8] Haruo Hosoya and Benjamin C Pierce. XDuce: A statically typed xml processing language. *Transactions on Internet Technology*, 3(2):117–148, 2003.

[9] Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. PADS/ML: A functional data description language. In *ACM SIGPLAN Notices*, volume 42, pages 77–83. ACM, 2007.

[10] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .NET Framework. In *Proceedings of the International Conference on Management of Data*, SIGMOD '06, pages 706–706, 2006.

[11] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Unifying tables, objects, and documents. In *Workshop on Declarative Programming in the Context of Object-Oriented Languages*, pages 145–166, 2003.

[12] Tomas Petricek, Gustavo Guerra, and Don Syme. F# Data: Making structured data first-class citizens. *Submitted to ICFP 2015*, 2015.

[13] Tomas Petricek and Don Syme. In the age of web: Typed functional-first programming revisited. *In post-proceedings of ML Workshop*, 2014.

[14] Stefan Scheglmann, Ralf Lämmel, Martin Leinberger, Steffen Staab, Matthias Thimm, and Evelyne Viegas. Ide integrated rdf exploration, access and rdf-based code typing with liteq. In *The Semantic Web: ESWC 2014 Satellite Events*, pages 505–510. Springer, 2014.

[15] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[16] Martin Sulzmann and Kenny Zhuo Ming Lu. A type-safe embedding of XDuce into ML. *Electr. Notes in Theoretical Comp. Sci.*, 148(2):239–264, 2006.

[17] Don 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. Strongly-typed language support for internet-scale information sources. Technical Report MSR-TR-2012-101, Microsoft Research, September 2012.

[18] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings of the Workshop on Data Driven Functional Programming*, DDFP'13, pages 1–4, 2013.

[19] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Not.*, 32(12):203–217, 1997.

[20] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.