Type providers

Authors
Affliations
emails@domain

Abstract

What is the problem?

Why is it important?

What do we do?

What follows from that?

1. Introduction

Modern applications do not run in vacuum. They execute in *information rich* environments, access data from relational databases, external files and from increasingly important web-based services. Stable, organized *information spaces* are now available via the web [5].

Despite this, few strongly-typed programming languages and tools are able to seamlessly integrate external information sources as if they were strongly-typed components from a programmer's perspective.

The recent version of F# introduced a *type provider* mechanism [5] which makes it possible to view diverse external data sources as statically typed components. However, this mechanism rises a number of concerns. Most importantly, if the type of the component depends on a schema of external data source, the type safety of program is relativized with respect to possible schema change of the data source.

TODO

- Give a unified model of *information spaces* using RDF triples and sketch how different data sources (including relational databases, XML and WorldBank) can be mapped to the triple model. We then distinguish between *schema* and *data* in the *information space*.
- We describe *type-bridging mechanism* – a formal model of F# 3.0 type providers – that specifies how to map an information space into a programming language. The mechanism takes some information from the *information space* as the *schema* and based on the schema, maps the information space into types and values. The choice of schema is reflected in the provided types.
- We formalize the notion of *relativized type safety* – informally, a type safe program using our type-bridging mechanism does not go wrong as long as the *schema* in the information space does not change.

Unpublished draft.

2. F# type providers

F# type providers [5] provide a bridging mechanism that exposes external data sources as statically typed components. Technically, type providers are compile-time components that provide the *signature* (a static type) of the external component together with its *implementation* (code that obtains data and exposes it in a format matching the signature).

The following example uses the Freebase [3] type provider to print a list of computer scientists:

```
#r "DataStore.Freebase.dll"
let data = DataStore.Freebase()
for c in data.Science.``Computer Scientists`` do
  printf "%s" c.Name
```

The type provider is referenced on the first line. It provides a type DataStore.Freebase that is constructed on the first line. The provided type has a property Science which returns an instance of another type with a property ``Computer Scientists``. The property has a type ComputerScientistsEntity list where ComputerScientistsEntity is another provided type with property Name (among many others).

The provider also provides an implementation of the types and properties. Using a type erasure mechanism (similar to the one used in Java Generics [2]), the ComputerScientistsEntity type is erased to a general FreebaseEntity type and the access to Name property is erased to a call c.GetStringValue("Name").

2.1 Questions

The example rises a number of questions – many of them are already answered in the F# type providers technical report [5]. In this paper, we focus on the following two themes.

Representation. To implement a type provider (e.g. for the Freebase data source), the developer needs to write F# code that connects to a data source and generates types based on the information available.

This mechanism is powerful, but it is difficult to analyse formally, because a type provider may execute arbitrary code. Thus our first aim is to find a simpler model that is more amenable to formal analysis (and might be also used to simplify the implementation of certain type providers).

Type safety. The above code is only correct as long as the Freebase schema does not change in certain ways. For example, renaming Name property to ``Full name`` would break the above code, because the Name is reflected at the type-level (and c.GetStringValue("Name") would fail because the value for the key "Name" is not available).

On the other hand, changing the name of "Tomas Petricek" to "Tomáš Petříček" is not a problem, because the change is only made in part of information that is available at value level. After such change, the code will print different results, but it will still work.

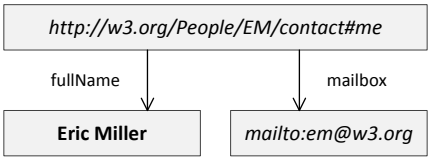

Figure 1. Graph representation of data

Thus our second aim is to analyse the type provider and specify what schema changes are safe and which are not. A type provider is type-safe with respect to a schema change if the schema change does not break the provided types and the provided implementation.

3. Information spaces

The F# type provider mechanism makes a reasonable assumption that there is no single universal schema language for describing *information spaces* – and thus it allows arbitrary F# code in the type provider.

However, to model type providers formally, we need to find a model of *information space* that is simple and amenable to formal analysis, but powerful enough to model a wide range of data sources accessible using type providers including hierarchical data (i.e. XML files), SQL databases, multi-dimensional data (such as Freebase, the World Bank data set) etc.

3.1 The RDF data model

Our formal model of information spaces is based on the Resource Description Framework (RDF), which *"is a language for representing information about resources in the World Wide Web"* [4]. RDF is based on the idea of modelling information using a graph. The nodes of the graph represent resources (identified using an URI) or primitive values and the nodes represent properties.

Figure 1 shows a basic example adapted from [4]. It shows a model with three edges and two nodes. One edge represents a primitive value (a string) and two other edges represent URIs (unique identifier of a person and email). The nodes associate values with the root edge and annotate them as fullName and mailbox.

The RDF model can be presented in form of *triples* consisting of *subject*, *predicate* and *object* (where subject and object are nodes and predicate is an edge in the graph). For the purpose of this paper and our model of type providers, we use a simplified formal model based on triples:

Definition 1. *Given a set of primitive values $\mathcal{P}$, a set of entity identifiers ? and a set of relation identifiers ?*

an information space *$(\mathcal{V}, \mathcal{R})$ is a triple of values $\mathcal{V}$ and relations between pairs of values $\mathcal{R} \in \mathcal{V} \times \mathcal{V}$. The relations are undirected, meaning that $(v_1, v_2) \in \mathcal{R} \Leftrightarrow (v_2, v_1) \in \mathcal{R}$.*

3.2 Examples

The previous simple definition is general enough to model a number of real-world data sources. We look at three examples showing how databases, XML and multi-dimensional data sets can be viewed as information spaces.

DB

XML

References

[1] D. Beckett, editor. *RDF/XML Syntax Specification (Revised)*.

[2] 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.

[3] Google. Freebase. `http://www.freebase.com/`, 2012.

[4] F. Manola and E. Miller, editors. *RDF Primer*. W3C Recommendation. World Wide Web Consortium, 2004. URL `http://www.w3.org/TR/rdf-primer/`.

[5] 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.