Improving Programmability of Linked Data Sources

Gerd Groener[1], Kenji Takeda[2], Don Syme[2], Ross McKinlay[2]

[1]Institute for Web Science and Technologies, University of Koblenz-Landau, Germany
[2]Microsoft Research Cambridge, UK

Abstract.

1 TODO: Proposals for title

Some ideas for the title:

- Towards a Data-oriented Type System in Programming Environments
- A type Bridging technique for RDF data sources
- Adapter Layer for Web data integration
- Connected Programming

2 Introduction

Publishing linked (open) data on the Web has become a success story during the last years. RDF as the underlying representation formalism allows for flexible modeling of data such that users are able to easily share, distribute and interconnect their own data all over the world. Publishing and representing data on the Web is one side of the coin, but writing programs and applications that access and smoothly integrate RDF data is another, rather challenging issue.

One key problem is the fundamental gap how data is processed in both areas. RDF representation benefits from the flexible data model, while software engineering environments and programming languages rely on powerful typing mechanisms that guide programmers in their development by intelligent IDE support and avoid run-time exceptions that might be caused by incompatible types.

The integration of large external data into programs and programming environments has already been investigated by dynamically typed programming languages. However, as it is in the nature of these languages, without type support when writing program code. Some initially approaches use adapter mechanism to integrate large data sources into statically typed programs. They rely on a rigit and well-known schema of the data source. Given such schema information, these approaches have led to good principles and toolins to allow for a smooth integration of data.

Looking into linked data, the question arises whether a smooth integration is even possible for RDF data sources. In particular, we are faced with the following problems. First, schema information might be partially unknown and even incomplete when data are accesses. Instead, the structure among individuals, which is given in terms of properties between individuals, might be used for a data access with respect to a certain structure. Second, while the schema is rather fix, the underlying data tend to change rather often. Thus, programs at run time might incorporate such aspects. Third, the sheer size of data sources (like DBpedia) might lead to a large number of types and accordingly to high type generation effort, while the program and system execution only need a part of the generated types. Thus, in an extreme case, this might even make the use of statically typed languages impossible.

In this paper, we present an integration and type bridge from RDF data sources to statically typed programming languages. We adopt the principles of *type provider* to define an adapter in terms af a library (i.e., a dll-file) for statically typed programming languages. This adapter can be loaded in a program to access an arbitrary RDF data source that offers a SPARQL endpoint.

3 What does programming Linked Data mean?

The linked data cloud consists of a variety of connected RDF data source, constituting a huge and steadily growing information space. In order to use these data in programs, an integration bridge is required such that the RDF data can be included in the program and type environment.

When writing programs and Web applications, programmers use programming languages and environments like IDEs the write their code. In statically typed programming languages, a powerful type system support programmers during the program design and ensure type correctness in program execution. In particular, statically typed programming languages provide the following benefits:

1. **Design Time Assistance:** The programming environment provides support programmers when writing code by context-sensitive auto-completion, interactive type checking (so called red squigglies) and quick information display like mouse-hover and detailed documentation (e.g., by pressing F1 button). All these interactive features help programmers in writing their code and give early feedback (at design time) about program correctness.
2. **Run Time Assistance:** When a statically program is compiled, type definitions and their usage in programs are checked. Thus, at run time, we can rely on a proper type system that offers features like type casts, type checking and type inference. Run time errors and exceptions during program execution based on incorrect type usage and type mismatch are already detected during design and compilation, and thus, might not cause errors at run time. Types can be even used to optimize type interpretations. comment Gerd: *I can't remember what the last sentence about optimize / drive interpretations mean.*

While these benefits of statically typed programming languages are obvious, the key question is how can such features be achieved when we access and integration RDF data sources, i.e., types in our type system refer to RDF classes and collections of individuals. This means, that types are used in the usual programming manner, for instance RDF individuals have all properties that are defined by the RDF classes. This is illustrated in the following example.

Assume a developer is programming an application for a mobile phone that accesses data about movies, retrieved from RDF data sources like DBpedia[1] via a SPARQL endpoint, e.g., as provided by DBpedia[2].

Step 1: Find a Class. First of all, the programmer decides to use DBpedia as a data source since it is well connected to other data sources. As a first step, the programmer has to look for a dedicated class for "movie" in the data source. For this purpose, the data source must be explored, e.g., by starting from a top RDF class and moving downwards the class hierarchy.

Problem: Thus, we need means to explore a data source and find the dedicated class.

Step 2: Define a Class / Type for a class. Once, a class for a "movie" is found, e.g., `http://dbpedia.org/ontology/movie`, the programmer has to define this class in the program.

Listing 1.1. Type Definition for RDF Class "Movie"

```

// The "Movie" Type
type Movie = {
  id : URI
  rdfs:label "Movie"
}
```

The *problem* we have to solve here is to map the RDF class description into a type definition in the program code.

Step 3: Define Related Types / Classes. Looking into the RDF class, we see that "Movie" is a subclass of "Work" (`http://dbpedia.org/dbpedia-owl/Work`). Hence, if we want to reflect this characteristic, we need to define a type for class "Work" as well.

Listing 1.2. Type Definition for RDF Class "Movie" and "Work"

```

// The "Work" Type
type Work = {
  id : URI
  rdfs:label "Work"
}

```

[1] DBpedia: `http://dbpedia.org`
[2] DBpedia SPARQL Endpoint: `http://dbpedia.org/sparql`

```
// Again the "Movie" Type as Subclass of "Work"
type Movie = {
  inherit Work
  id : URI
  rdfs:label "Movie"
}
```

Obviously, besides the "Work" class, other class might be created as Movie is related to them. This procedure might even continue since these other classes like "Work" can depend on other classes.

Given the sheer size of linked data sources (or even the linked data cloud), the key *problem* is which classes need to be integrated in a program, i.e., for which RDF classes is a type definition necessary. Building types for all classes of a data source is definitely not scalable, ane even not needed since a particular application might only a part of the classes. Thus, the question is whether it is possible to build types only *on demand*.

Step 4: Define individuals of classes. In our application, we want to mange concrete movies, which can be derived from the data sources, as individuals of this class. Using a SPARQL query we can retrieve all individuals of class "Movie". For instance, we get the individual `http://dbpedia.org/page/Skyfall` for the movie "Skyfall". Accordingly, we can build an instance of movie.

Listing 1.3. Individual of "Movie"

```
  let skyfall = Movie('http://dbpedia.org/page/Skyfall')
```

Step 5: Incorporate Properties of Individuals. It is in the nature of the flexible RDF model that properties can be defined for individuals without explicit definition of these properties for classes.

3.1 Contribution

We distinguish between challenges and our corresponding contributions that are related to (i) the programming environment, (ii) the mapping from schema and data requests to SPARQL queries and (iii) the Semantic Web and linked data oriented investigations.

Programming Language and Environment

We present a type bridge / adapter in order to create types for RDF classes that are retrieved from linked data sources. In particular, this includes the following aspects.

comment Gerd: *Here we need a description of type provider features.*

Type Integration and Type Inference. When deriving RDF data from external data sources, the key problem is how to integrate such types into the host programming schema.
Contribution: Types are built based on the schema information that is obtained from the RDF data source.

Type Definition on Demand — Scalability. As data sources on the Web tend to be huge, it is not a promising idea to build the types for all classes of a data source. Obviously, types are only needed if particular applications need to access them. **Contribution:** on demand typing based on the current element.

Incorporate Data Changes. It is obvious that RDF data change rather frequently, while the schema remains stable. Thus, it is meaningful to build types wrt. schema (at design time) and populate these types at run time.
Contribution: Types are built wrt. the schema (class definitions in RDF). Classes are populated by individuals at run time. This implicitly also takes changes of the data (individuals) into account.

Semantic Web and Linked Data

While our type bridge is built to access and integrate RDF data into programs, we also use Semantic Web technologies and built the data access upon these existing means. In particular, we apply SPARQL queries, the SPARQL entailment regime, which includes RDF(S) entailment, and we actually rely on best practices for publishing linked data.

Derive fine-grained Schema from RDF Data. Hierarchies of classes and also properties in RDF data can be quite extensive. Besides this, domain and range restrictions of properties that entities can be classes in case this is not explicitly stated.
Contribution: We incorporate RDF entailment regime, which is supported by SPARQL 1.1 in order to derive a fine-grained type system / schema.

Navigation on Class vs. Instance level. In RDF data sources, property specifications at the class level are often rare, for instance in DBpedia classes have only three or four properties and these are actually quite generic one, derived from super-classes. Instead, the most interesting way for navigating is at the instance level. But, how can we cover properties if their corresponding class in the programming language does not have this property. We can even not assume this property for the class since the individuals do not necessarily share their properties.
Contribution: We offer a two-layered navigation in RDF sources

Property-based Classes. RDF data have type statements that assign data (individuals) to classes. However, it has been shown that besides the type statements properties of individuals are an essential means to group individual to a kind of classes instead of only relying on explicitly specified types.
Contribution: We allow the specification of types as a set of properties.

Set-based Classes. Following the OWL-based set semantics, classes in the RDF sense can be considered as unions or intersections of RDF classes.
Contribution: We allow the specification of types as a unions and intersection of existing types.

3.2 Semantic Web and Linked Data

comment Gerd: *not sure, whether the mapping to SPARQL is worth to mention.*

4 Foundations

An RDF[3] data source consists of at least one RDF graph, which is a set of RDF triples (s, p, o) that consists of subject (s), predicate or (p) and object (cf. Def. 2).

Definition 1 (RDF Graph). *Let U be a set of URIs, L a set of literals and B a set of blank nodes, with $U \cap L \cap B = \emptyset$.*

An RDF graph is defined as: $\mathrm{G} = \{(s,\ p,\ o) \in (U \cup B) \times U \times (U \cup L \cup B)\}$.

SPARQL[4] is a query language for RDF graphs with select, from and where clauses. Like an RDF graph, the graph pattern of the where clause consists of RDF triples, in which variables are allowed as subjects, predicates and objects of triples. The result of a query is a binding of the variables in the select clause, while the binding is determined by matching of triples from the where clause to triples in the RDF graph G.

In SPARQL 1.1, which we are referring to in this paper, the graph matching principle between triples in the query and the data source is extended by entailment relations, as defined by the SPARQL entailment regimes[5]. Among others, the entailment regimes contain RDF and RDFS entailment rules Thus, triple matching is extended to triples that can be derived from an RDF graph G. Formally, we denote this extended set of triples, which can be derived by RDF(S) entailment as Materialized Grapn (cf. Def. **??**). (The symbol $\models_{\mathcal{T}}$ denotes RDF(S) entailment.)

Definition 2 (Materialized RDF Graph). *Let* G *be an RDF graph. A materialzed graph $\hat{G}$ is defined as follows:*

$\hat{G} = \{(s,\ p,\ o) \in (U \cup B) \times U \times (U \cup L \cup B) | G \models_{\mathcal{T}} (s,\ p,\ o)\}$.

Foundations: F# and Type Provider

[3] RDF Primer: `http://www.w3.org/TR/rdf-primer`
[4] SPARQL 1.1 Query Language: `http://www.w3.org/TR/sparql11-query`
[5] SPARQL 1.1 Entailment Regimes: `http://www.w3.org/TR/sparql11-entailment`

5 A Type Bridge for RDF

comment Gerd: *not sure we use 'bridge', 'adapter' or 'type provider'?*

Our goal ist to build an integration bridge for arbitrary RDF data sources with a SPARQL endpoint in order to access and integrated RDF data into a statically typed program on demand. When using these bridge, it should not be part of the actual developed application, instead it is just used as a library. In the following, we will see some technical details how the bridge is developed.

5.1 Life Cycle: Program Design Time and Run time

explain the principle: (i) writing a TP, (ii) using a type provider

Our type bridge is a compile time component. It has static parameters like the address (URI) of the SPARQL endpoint and optional parameters like the number of individuals that should be retrieved. After a successful connection to the SPARQL endpoint, we get the following artefacts for data access and connection:

1. The *signature* of the data source, which is given by classes and properties in the data source, is retrieved. comment Gerd: *not really retrieved — can be retrieved*
2. The *type creating* on demand.

5.2 Type Definitions

The type bridge consists of two components: (i) the data source *connector* and (ii) the *type generation description*. The latter one is a kind of compile-time meta-programming construct.

- Define provided classes
- define provided properties
- subclass navigation

6 Type Provider — Usage and Features

What are the concrete features here:

- design time: intellisense, autocompleting, when designing types
- run time: lazy typing

References