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 RDF data model is basically a graph, which is constitued of a set of subject-predicate-object triples. This graph covers both the schema in terms of classes and the data in terms of individuals The usage of certain predicates like rdf:type, rdfs:domain and rdfs:range and classes like rdf:Class indicate that certain entities in the graph are classes. Thus, this is the only evidance for a schema.

3.1 Examplified Problem Description

When writing programs and Web applications, programmers use programming languages and environments like IDEs the write their code. Programming languages typically rely on some kind of schema or types where types (or classes) serve as containers to manage data. The organization of types in a so-called type system is an essential benefit of *statically typed* languages where 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. In each step, we see the corresponding requirement that has to be addressed by a type bridge. We can distinguish between *retrieval requirements (RR)* and *typing requirements (TR)*.

Assume a developer is programming an application for a mobile phone to provide additional data for movies like information about actors or genre. These additional data are retrieved from RDF data sources like DBpedia[1] via a SPARQL endpoint, e.g., as provided by DBpedia[2]. These data should be included in the application on demand, i.e., only if needed and they should be managed in the application in a typed fashion, as described above.

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 "Actor" in the data source. For this purpose, the data source must be explored, e.g., by showing all RDF classes in a list such that the developer can select the class of interest.

RR 1: Thus, we need means to explore a data source in terms of retrieving classes in RDF data sources.

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

Listing 1.1. Type Definition for RDF Class "Movie"

```
1
2 // Type definition for "Actor"
3 type Actor = {
4   id : URI
5   rdfs:label : String
6 }
```

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

TR 1: 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. Looking into the RDF class, we see that "Actor" is a subclass of "Artist" (`http://dbpedia.org/dbpedia-owl:Artist`). Hence, if we want to reflect this characteristic, we need to define a type for class "Artist" too.

Listing 1.2. Type Definition for RDF Classes "Actor" and "Artist"

```
1
2  // The "Artist" Type
3  type Artist = {
4    id : URI
5    rdfs:label : String
6  }
7
8  // Again the "Actor" Type as Subclass of "Artist"
9  type Actor = {
10   inherit Work
11   id : URI
12   rdfs:label : String
13 }
```

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

TR 2: 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 "Actor". For instance, we get the individual `http://dbpedia.org/page/Bruce_Lee` for the actor "Burce Lee". Accordingly, we can build an instance of movie.

Listing 1.3. Individual of "Actor"

```
1 let bl = Acotor('http://dbpedia.org/page/Bruce_Lee')
```

TR 3 Types are used to instantiate individuals.

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. For instance, the individual "Burce Lee"

has properties that are not specified for stated as properties for the RDF class "Actor".

TR 4 and **RR 2**: Incorporating properties of individuals at the type level. From a programming language perspective, this is a rather callenging issue since a type usually specifies properties, which can be even optional, but in our case, the previously sketched type specification does not even know about additional properties.

3.2 Contribution

We distinguish between 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.

We present a type bridge or type adapter in order to create types for RDF classes that are retrieved from linked data sources. In essence, the type bridge acts as a type provider in F# that support the integration of information sources into F# [1]. Type provider are means for data progamming in statically typed programming languages. The general principle of type providers is to retrieve a well structured schema from (Web) data sources in order to build the corresponding types at run-time.

In our case, we can not rely on a well structured schema, thus we have to incorporate RDF modeling characterisits in order to integrate data into a static type system. The closed related type bridge is the Freebase type provider that allows for the navigation within the graph-structure of Freebase [3].

The RDF type bridge includes the following aspects.

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. The schema is derived by the usage of certain predicates like rdf:type, rdfs:domain and rdfs:range and classes like rdf:Class.

Scalable Type Definition on Demand. 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

[3] The Freebase Wiki about the Schema: `http://wiki.freebase.com/wiki/Schema`

changes of the data (individuals) into account.

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. 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.

Retrieval 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.

4 Foundations

An RDF[4] 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[5] 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[6]. Among

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

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.)

We use a distinction between classes (schema) and data in the RDF graph. We refer an entitiy C (subject or predicate in a triple) as a class (or RDF class). This is described by the second part of Def. 2.

Definition 2 (Materialized RDF Graph and RDF Classes).

1. *Let* G *be an RDF graph. A materialzed graph* $\hat{G}$ *is defined as follows:* $\hat{G} = \{(s,\ p,\ o) \in (U \cup B) \times U \times (U \cup L \cup B) | G \models_{\mathcal{T}} (s,\ p,\ o)\}$.
2. *C is referred to as a class (or RDF class) if one of the following conditions hold: (i) there is a type statement (s rdf:type C)* $\in \hat{G}$*, (ii) there is a domain restriction (s rdfs:domain C)* $\in \hat{G}$ *or (iii) there is a range restriction (s rdfs:range C)* $\in \hat{C}$ *and C is not a datatype.*

Foundations: F# and Type Provider

Definition 3 (Type System). *A type system is a tuple* $(\Gamma, \mathcal{T})$ *Let* Γ *be a domain,* $\mathcal{T}$ *a set of type expressions (or types for short).* $\top \in \mathcal{T}$ *is the most general type expression that is a super-type of all other types.*

For each element in the domain $e \in \Gamma$ *there is a type* $\tau \in \mathcal{T}$ *such that e is an individual of type* τ*:* $e : \tau$

Definition 4 (Class-Type Mapping). *An RDF graph G is mapped to a type system* $(\Gamma, \mathcal{T})$ *by a class-type mapping m. Each (named) RDF class C is mapped to a type* $\tau \in \mathcal{T}$*. Each property is mapped to a type tuple* $(\tau_1, \tau_2) \in \mathcal{T} \times \mathcal{T}$.[7]

5 A Type Bridge for RDF

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

comment Gerd: *we have to make the distinction clear: (i) the work we did (meta-level) and (ii) how the TP is used as a library (run time)*

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.

[7] Note: If the domain or range class of a property is unknown, the corresponding types are $\top$.

5.1 Features of the Type Bridge

According to Sect. 3, we aim at achieving two groups of requirements: navigation requirements (NR) and typing requirements (TR). Navigation is needed since the an RDF graph does not explicitly distinguish between schema and data, as for instance in relational databases. Thus, the navigation is used to find those elements in the graph that should be integrated as types and their corresonding data (individuals) in a program. Aspects of the integration into the programming type system are reflected by the typing requirments (TR).

In order to meet the requirements, the type bridge has to provide the following functionality. First, the exploration of an RDF graph must be supported. Thus, we need to be able to hook into the graph and then follow the properties from one entity to the next one. While this is possible for RDF classes and individuals. Second, the retrieval of the currently explored entity and its properties must be supported.

The type bridge, its features and its context of use is illustrated in Fig. 1. The design and implementation of the type bridge is described in the next part of this section.

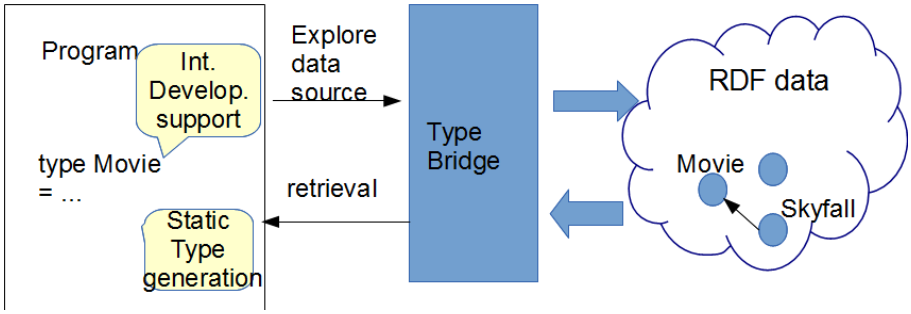

Fig. 1. Functionality and Context of the Type Bridge

5.2 Type Definitions

Our type bridge is a compile time component. It has static parameters like the address (URI) of the SPARQL endpoint. 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. In the following, we consider both components in detail.

The connector establishes a connection to the SPARQL endpoint. It serves as a mediator between the type generation descrition and the external data source. The connector gets as an argument the URI of the SPARQL enpoint in order to establish a connection to the SPARQL endpoint.

The connector consists of functions that are used by the type generation description to navigate in the data source and to retrieve data. Internally, these functions encode SPARQL queries to implement both tasks the navigation and the retrieval.

The type generation description defines how types with their parameters, properties and methods are built.

The first part is the definition of the type itself, which requires at least the name of the type and its erasure, i.e., the type this type can be erased to.

Let C be the class, which is retrieved by a SPARQL query. The type generation description in F# is as follows:

Listing 1.4. Type Rule: Type Generation Description for RDF Class C

```
1 t = ProvidedTypeDefinition(cName, baseType=Some typeof<obj>)
```

In Listing 1.4, `cName` denotes the name of class C, e.g., the URI of C. The `baseType` gives the "super-type" from which the generated type t inherits from. As an example, assume we specify a type generation description for RDF classes. It description, which is a compile-time meta-programming constuct, is used to build RDF classes, e.g., the "Movie" class in our example.

Types can have properties, accordingly if a type will be created, the corresponding properties have to be created too. These properties are obtained from the RDF data source and should be directly reflected in the source code that will specify the type. Accordingly, the type generation description describes how to obtain properties and to generate the properties of these types.

Listing 1.5. `Property Rule:` Add Property for Class C

```
1 let p = ProvidedProperty(pName, typeof<string>,
2    GetterCode = (fun args ->
3                     <@@(%%(args.[0]):obj) :?> string  @@>))
4  // add this property (delayed)
5 t.AddMemberDelayed(p)
```

In Listing 1.5, `pname` denotes the name of the property (e.g., the URI), `typeof` gives the super-tye (string in this case) and `GetterCode` is a so-called quotation that specifies the result of a property selection at run time. In this simple case, the string representation of the property itself is the meaning of the property. In the second line, the peroperty is "added" to the type. Note, this this depends on the concrete generated type when the bridge is used in a concrete program. For instance, if a type for RDF class "Actor" is created (according to the rule in Listing 1.4), the properties (predicates in the RDF graph) of "Acotr" are generated and added (according to the property rule in Listing **??**).

Finally, individuals can be added to classes according to Listing 1.6. For this, a new type would be created with the name of the class (denoted by `cName`) and the suffix "Individuals". This type is added to t. In this collection, all individuals are contained.

Listing 1.6. `Individual Rule:` Add Individuals (set / collection) to Class C

```
let individuals =
   ProvidedTypeDefinition(cname+"Individuals", Some typeof<obj>)
\\ add these individuals
individuals.AddMembersDelayed()
\\  add indiviual collection to type
t.AddMemberDelayed(indiviuals)
```

5.3 Additional Constructs

According to the integration principle, types are created when a certain entity is retrieved from the graph. While this entity is reached by navigation along the graph. Thus, we can also have some cycles in the exploitation. In order to achieve skalability in statically typed languages, it has to be ensured that a type for entity is only created once. To do this, we use dictionaries in the type dictionaries (hash maps) to ensure that types are unique and to avoid infinite loops.

Another technique that we apply is a sampling of properties. According to requirements **TR 4** and **RR 2**, the a typed access on individuals suffers from the problem that the type definition (based on the RDF class) to not necessarily reflect all property statements of the individuals. To remedy this, we apply a sampling technique when a class type is created. A static parameter of the type bridge gives the number of sample individuals. The properties of these individuals are added as optional properties of the corresponding types. Further parameters allow to specify how often a property must appear in order to be added as property to a type.

6 Linked Data Programming — Type Bridge in-use

The type generation definition, as presented in Sect. 5, can be applied for arbitrary RDF data sources. The usage in two steps. First, when writing a program, e.g., an application for a mobile phone (cf. Sect. 3), gets the full support from the underlying static type system. Second, when the application is used at run-time, these generated types are treated as basic .NET types in F#.

Consider again the program development from the original example in Sect. 5. When developing a progam by using the RDF type bridge, the libarary of the bridge (dll-file) must be loaded and then a data handler is created, which invokes the connector component to establish a connection to a particular RDF source, e.g., to DBpedia. Afterwards, the data handler serves as an object in the program to retrieve and organize retrieved data during the development.

6.1 Integrated Data Retrieval

The retrieval of RDF data is completely integrated in the IDE of the programming language, i.e., Visual Studio[8] or MonoDevelop[9] (for Linux systems) forF#

[8] Visual Studio: `http://www.microsoft.com/germany/visualstudio/`
[9] MonoDevelop IDE: `http://monodevelop.com/`

in our case. Thus, a developer can explore an external RDF data source and retrieve data from this source within the development environment and by using the data handler. Accessing the data handler is done by using the usual dot-operator.

Accordingly, "accessing" the data handler by the dot-operator is the IDE-integrated access for the navigation and retrieval of data from the external RDF data source. In particular, using the dot-operator depends from the current status of the data handler.

- After the instantiation of the data handler, it refers just to the RDF data source, e.g., DBpedia. This is represented by a root type (also called service type) `http://dbpedia.org`. When pressing the dot we get a list of all classes of the data source is shown.
- After selecting one of the classes, the status of the data handler refers to this class. For instance, if the developer selects the RDF class "Actor" (from the list of classes shown by the data handler), the current status of the data handler is like an "Actor" class. In the program, this status is represented by the path `<http://dbpedia.org/>.<http://dbpedia.org/ontology/Actor>`.
- When the status of the data handler is a particular class, the dot-operator will list all properties of the class and a collection called "Individuals" that contains all individuals of this class.
  - When a property is selected, the status of the data handler is this particular selected property. For instance, for Actor we can select the property "rdfs:subClassOf". Using the dot-operator get a list of classes that are the super-classes of the domain-class. For instance, the class "Artist". When one class of this list is selcted, the status of the data handler is like in the second case a certain RDF class.
  - When the individuals collection is selected, the develop gets a list of individuals. If one is selected, e.g., "Bruce Lee", the status of the data handler is the current individual. When using the dot-operator for an individual state, e.g., for "Bruce Lee", the developer can see a list of properties, but now these are all properties of this particual individual.

7 Evaluation

8 Related Work

Related work can be disinguished into two groups: the exploration of data sources (data access) and the integration into programming environments.

Several related approaches study the exploration and visualization of Web data sources. The aim of this work is to allow users without SPARQL experiences an easy means to get information from linked data sources. tFacet [2] and gFacet [3] are tools for faceted exploration of linked data sources via SPARQL enpoints. fFacet provides a tree view for navigation, while gFacet has a graph facet for browsing. The navigation of RDF data for the purpose of visualizing

parts of the data source is studied in [4], but with the focus on visualization aspects like opitimization of the displayed graph area. In contrast our work, these approaches do not consider any kind of integration aspects like code generation and typing. Furthermore, the navigation is rather restricted to a simple hierarchical top-down navigation.

Data[10] provides a protocol to query and update external data via RESTful service. OData contains a data model, logically based on an ER model, such that the protocol can describe both data (called entities) and types (entity types). Some implementations provide access from OData applications to SPARQL endpoints[11].

References

1. Syme, D., Battocchi, K., Takeda, K., Malayeri, D., Fisher, J., Hu, J., Liu, T., McNamaa, B., Quirk, D., Taveggia, M., Chae, W., Matsveyeu, U., Petricek, T.: F# 3.0 — Strongly Typed Language Support for Internet-Scale Information Sources. Technical Report MSR-TR-2012-101, Microsoft Research (2012)
2. Brunk, S., Heim, P.: tFacet: Hierarchical Faceted Exploration of Semantic Data Using Well-Known Interaction Concepts. In: International Workshop on Data-Centric Interactions on the Web (DCI 2011). Volume 817 of CEUR-WS.org. (2011) 31–36
3. Heim, P., Ziegler, J., Lohmann, S.: gFacet: A Browser for the Web of Data. In: International Workshop on Interacting with Multimedia Content in the Social Semantic Web (IMC-SSW 2008). Volume 417 of CEUR-WS. (2008) 49–58
4. Dokulil, J., Katreniaková, J.: Navigation in RDF Data. In: 12th International Conference on Information Visualisation, IEEE Computer Society (2008) 26–31

[10] OData (Open Data Protocol): `http://www.odata.org/`

[11] OData SPARQL: `https://github.com/BrightstarDB/odata-sparql` last visit 2013-05-07