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. Linked data sources benefit from the flexible data model of RDF that can easily be extended and does not have a fixed schema. While this representation facilitates the publication and distribution of linked data on the Web, it complicates the integration and processing of data from these sources in programming environments since programs usually rely on types and schema information to internally organize their data. In this paper, we present the design and implementation of an RDF type bridge, which is a compile-time component and acts as a programming interface for arbitrary external RDF data sources. As a result, programmers can use the type bridge to write programs that access and integrate external RDF data. By loading the bridge into a program, the type bridge offers development support by an interactive exploration of the data source and ensures strong typing of retrieved RDF data.

1 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 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 (cf. [1]) use adapter mechanism to integrate large data sources into statically typed programs. They rely on a rigid and well-known schema of the data source. Given such schema information, these approaches have led to good principles and powerful tooling to allow for a smooth integration of external 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* and define an adapter as a kind of programming interface for statically typed programming languages to RDF data sources. The resulting type bridge can be loaded into programs as library in order to access an arbitrary external RDF data source, which offers a SPARQL endpoint. The type bridge is used to (i) retrieve data from the RDF data source and (ii) to statically type the retrieved data and integrate these generated types into the type system of the host programming language. The type bridge is implemented in `F#`.

2 What does Programming Linked Data Mean?

The RDF model is a graph, which is constituted of a set of subject-predicate-object triples. This graph covers both the schema in terms of classes and the data by 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 evidence for a schema.

2.1 Exemplified Problem Description

When writing programs and Web applications, programmers use programming languages and environments like IDEs to 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 supports 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 supports 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 the 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.

While these benefits of statically typed programming languages are obvious, the key question is how can such features be achieved when we integrate RDF data, 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 from 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 a corresponding type in the program.

Listing 1.1. Type Definition for RDF Class “Movie”

```

// type definition for "Actor"
type Actor = {
  id : URI
  rdfs:label : String
}
```

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

TR 1: The problem that we have to solve here is to map the RDF class description into a type definition in the program code. This should be done automatically.

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, we need to define a type for class "Artist" too.

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

```

// type definition for  "Artist"
type Artist = {
  id : URI
  rdfs:label : String
}
// type definition for "Actor"  as subtype of "Artist"
type Actor = {
  inherit Artist
  id : URI
  rdfs:label : String
}
```

Obviously, besides the "Artist" class, other class might be created since the actor is related to them. This procedure might even continue since these other classes like "Artist" can depend on further 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, and even not needed since a particular application might only require 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 manage concrete actors that 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 "Bruce Lee". Accordingly, we can build an instance of movie.

Listing 1.3. Individual of "Actor"

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

TR 3: Types are used to instantiate individuals. In this case, properties of the individuals are specified at the type level such that accessing these properties at the individual level conforms to the type characteristics.

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

TR 4 and **RR 2**: Properties of individuals have to be incorporated by types. From a programming language perspective, this is a rather challenging 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, and such a situation has to be avoided in typed programming languages.

2.2 Features of the Type Bridge

The aim of the RDF type bridge (or RDF type adapter) is 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 supports the integration of information sources into F# [1]. Type providers are means for data programming 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 characteristics in order to integrate data into a static type system. The closest related type bridge is the Freebase type provider that allows for the navigation within the graph-structure of Freebase [3]. The RDF type bridge contains the following additional 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.
Feature: 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. **Feature:** On demand typing based on the current element. Types are created lazily. To to this, we need an efficient management of the already created types in order to avoid unnecessary type creating, which might be possible due to cycles in the RDF 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.
Feature: We incorporate RDF entailment regime, which is supported by SPARQL 1.1

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

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.

Typing of Classes base on Instance Properties. 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.
Feature: When defining a type for an RDF class, the type bridge offers two possibilities. The basic principle is to add properties to the type definition that are obtained as properties of the corresponding RDF class. As an addition, properties that are only on the individual level are also specified at the class level. This is done by sampling individuals of an RDF class and then choose the most common properties. Both the sample size and the threshold for "popular" properties are given as parameters.

3 Foundations

An RDF[4] data source contains at least one RDF graph, which is a set of RDF triples (s, p, o) that consists of subject (s), predicate (p) and object (o) (cf. Def. 1).

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: $G = \{(s,\ p,\ o) \in (U \cup B) \times U \times (U \cup L \cup B)\}$.*

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 others, the entailment regimes contain RDF and RDFS entailment rules Thus,

[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`

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 Graph (cf. Def. 2.A). (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 entity C (subject or predicate in a triple) as a class (or RDF class). This is described in Def. 2.B.

Definition 2 (Materialized RDF Graph and RDF Classes). *Let* G *be an RDF graph.*

A. A materialized graph $\hat{G}$ is defined as follows:
$\hat{G} = \{(s,\ p,\ o) \in (U \cup B) \times U \times (U \cup L \cup B) \mid G \models_{\mathcal{T}} (s,\ p,\ o)\}$.
B. C is referred to as a class (or RDF class) if at least 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) $\in \hat{G}$ and C is not a data-type.

Types classify the kinds of individuals (or objects) that are manipulated in a program or application. A type system is a syntactic technique to classify statements in order to avoid unintended program behavior. We consider a type system as specified in Def. 3.

Definition 3 (Type System). *Let U be a set of URIs, L a set of literals. A type system consists of a domain Γ, a set of entities M ($M \subseteq \mathcal{P}(U \cup L)$) and type expressions (or types for short) τ. The statement $\Gamma \vdash M : \tau$ means that entity M has type τ in a domain / environment Γ.*

4 A Type Bridge for RDF

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.

4.1 Features of the Type Bridge

According to Sect. 2, we aim at achieving two groups of requirements: retrieval requirements (RR) and typing requirements (TR). Retrieval is needed since an RDF graph does not explicitly distinguish between schema and data, as in relational databases. Thus, the retieval 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 requirements (TR).

In order to meet these requirements, the type bridge has to provide the following functionality. First, the exploration of an RDF graph must be supported.

Thus, we need some kind of data handler to be able to hook into the RDF source. Second, we have to use the data handler to explore the data source by following the properties from one entity to the next one. This must be provoded for RDF classes and individuals. Properties also include type statements, and therefore, we can also follow relations between classes and individuals. Third, the retrieval of the currently explored entity and its properties must be supported. Fourth, The definition of a static type must be able. Thus also all properties of the type must be defined.

The type bridge, its features and its context of use is illustrated in Fig. 1. The type bridge consists of a connector and a type generation description. The bridge uses SPARQL to connect to an RDF data source. The program, which uses the type bridge, has to use a data handler to get data from the data source and to finally build the types for retrieved data.

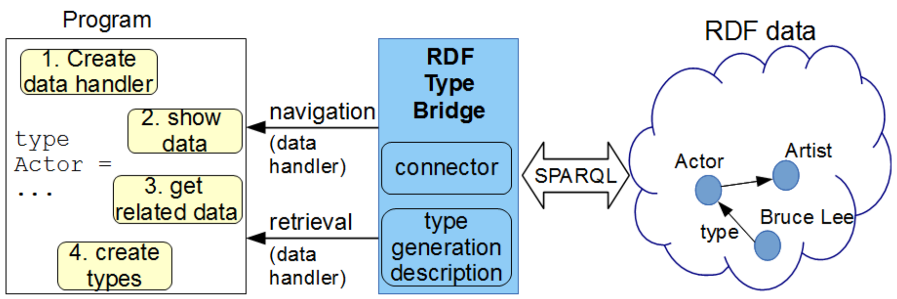


Fig. 1. Functionality and Context of the Type Bridge

4.2 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.

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

Functions of the connector are used to retrieve (i) all classes of the source, (ii) individuals of a certain class, (iii) properties of a class (i.e., the objects of triples in which the class is the subject and the property the predicate), (iv) properties of an individual and (v) the classes of an individual. The notion of a class in the RDF data referes to the specification of Def. 2.

The key part of the type bridge is the *type generation description* that defines how types with their parameters, properties and methods are built when the bridge is used. In essence, the type bridge is a model that specifies the type import into a language. This type import is specified by a set of rules.

Before we discuss these rules, we present one basic construct, the data handler or data context provider, of the type bridge that is used by programs to work with the type bridge. The type bridge itself specifies one base type, let's call it `rootType`. This type is created whenever a program uses (and therefore instantiates) the type bridge. The corresponding specification is shown in Listing 1.4. The argument ns denotes the namespace.

The `rootType` has one method (line 3, 4 in Listing 1.4) that returns the data handler, which is later used in the program environment to handle the connection to the type bridge. The `InvokeCode` specifies that when the method `getDataHandler` is called, the `Connector`, according to the above description, is intialized. The argument is the URI of a SPARQL endpoint (e.g., the endpoint URI of DBpedia).

Listing 1.4. Basic Rule: For the generation of a root type with data handler

```
1 let rootType = ProvidedTypeDefinition(ns, rootTypeName,
2    baseType=Some typeof<obj>)
3 let m = ProvidedMethod("GetDataContext", [],
4     InvokeCode = (fun _ -> <@@ Connector(sourceUrl) @@>))
```

In the following, we describe rules for building type definitions. The first step is the definition of the type for RDF classes, which can be used in programs to define types for RDF classes from the data source. The type definition 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 that is retrieved by a SPARQL query. The type generation description in F# is as follows:

Listing 1.5. Type Rule: Type Generation Description for RDF Class C

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

In Listing 1.5, `cname` denotes the name of class C, e.g., the string of the URI of C. The `baseType` gives the erased type, a"super-type" from which the generated type inherits. For instance, an RDF class can just be erased to an F# object type. If a type should have other features like being iterable, the type can be erased to sequence or list and thus, this type can serve as a collection element and iteration features on its elements will part of the type characteristics.

This type rule would be used if a programmer specifies a description for an RDF class. This description, which is a compile-time meta-programming constuct, is used to build the concrete RDF class, e.g., the “Actor” 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 corresponding type. Accordingly, the type generation description describes how to obtain properties and to generate the properties of these types.

Listing 1.6. Property Rule: Add Property for Class C

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

In Listing 1.6, `pname` denotes the name of the property (e.g., the URI), `typeof` gives the super-type (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. Such a provided property is created for each property of the RDF class C. In the second line, the peroperty is lazily “added” to the type. Here, type symbol t referes to the type of RDF class C. Note, the added properties depend on the concrete generated type when the bridge is used. For instance, if a type for RDF class “Actor” is created (according to the rule in Listing 1.5), the properties (predicates in the RDF graph) of “Actor” are generated and added (according to the property rule in Listing 1.6).

In the next rule, we treat individuals. They can be added to their corresponding classes. The according type generation description is shown in Listing 1.7. 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. We can see that the erased type (base type) of the individuals peroperty is a sequence (seq).

Listing 1.7. Individual Rule: Add Individuals (set / collection) to Class C

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

4.3 Additional Constructs

Types are created when a certain entity is retrieved from the graph, while this entity (e.g., a class) is reached by navigating along the graph. Thus, we can also have some cycles in the exploitation. In order to achieve scalability in statically typed languages, we have to ensure that a type for an entity is only created once. To do this, we manage types in dictionaries i(hash maps) to ensure that types are unique.

Another technique that we apply is the sampling of properties. According to requirements **TR 4** and **RR 2**, 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 whenever a class type is created. A static parameter of the type bridge gives the number of sampled individuals. The properties of these individuals are added as optional properties to the corresponding types. Further parameters allow to specify how often a property must appear in order to be added. The rule for creating a property (Property Rule), as shown in Listing 1.6, is the same, but it considers also these additional properties that are sampled from the individuals.

5 Linked Data Programming — the Type Bridge in-use

The type bridge, as presented in Sect. 4, can be applied for arbitrary RDF data sources. The contribution of the type bridge is twofold. First, when writing a program, e.g., an application for a mobile phone (cf. Sect. 2), the programmer gets the full support from the underlying static type system. This includes programming environment or IDE support like auto-completion, interactive type checking and error highlighting. Second, when the application is used at runtime, these generated types behaves as basic .NET types in F#.

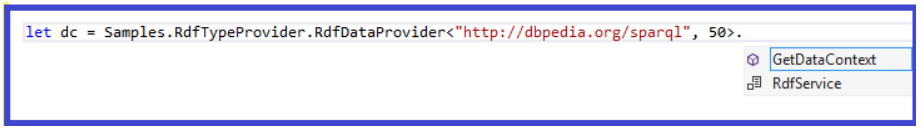
Consider again the program development from the original example in Sect. 4. When developing a progam by using the RDF type bridge, the libarary of the bridge (dll-file) must be loaded and then a data handler is created, which invokes the connector component to establish a connection. Afterwards, the data handler serves as an object in the program to retrieve and organize the retrieved data during the development.

5.1 Integrated Data Retrieval

The retrieval of RDF data is completely integrated in the IDE of the programming language, i.e., Visual Studio[7] or MonoDevelop[8] (for Linux systems). Thus, a developer can explore an external RDF data source and retrieve data from this source within the development environment by using the data handler. Accessing the data handler is done by using the usual dot-operator. The data handler is obtained by calling the `GetDataContext()` method as outlined in Fig. 2.

Accordingly, "accessing" the data handler by the dot-operator is an IDE-integrated access for the navigation and retrieval of data from the external RDF

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

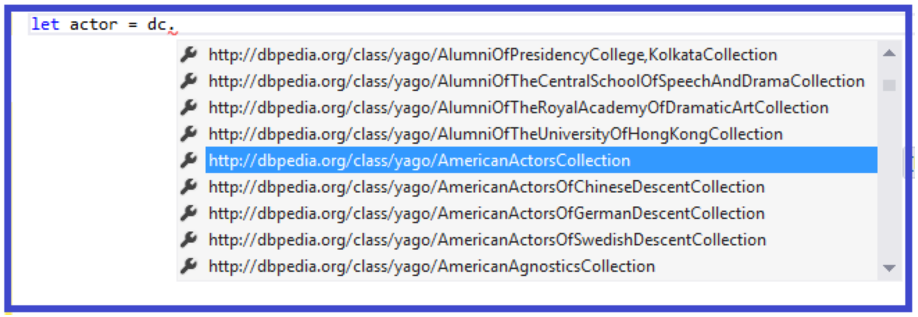


```
let dc = Samples.RdfTypeProvider.RdfDataProvider<"http://dbpedia.org/sparql", 50>.
```

Fig. 2. Methode `GetDataContext()` to create the data handler

data source. In particular, using the dot-operator depends from the current *status* of the data handler.

1. 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. This is illustrated in Fig. 3.



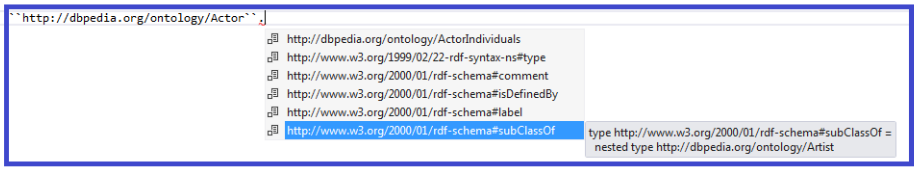
```
let actor = dc.
```

http://dbpedia.org/class/yago/AlumniOfPresidencyCollege,KolkataCollection
http://dbpedia.org/class/yago/AlumniOfTheCentralSchoolOfSpeechAndDramaCollection
http://dbpedia.org/class/yago/AlumniOfTheRoyalAcademyOfDramaticArtCollection
http://dbpedia.org/class/yago/AlumniOfTheUniversityOfHongKongCollection
http://dbpedia.org/class/yago/AmericanActorsCollection
http://dbpedia.org/class/yago/AmericanActorsOfChineseDescentCollection
http://dbpedia.org/class/yago/AmericanActorsOfGermanDescentCollection
http://dbpedia.org/class/yago/AmericanActorsOfSwedishDescentCollection
http://dbpedia.org/class/yago/AmericanAgnosticsCollection

Fig. 3. Use dot-operator direclty on the data handler

2. 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 an “Actor” RDF class. At this point in time, the type for Actor is not created, only the status of the data handler refers to a certain RDF class,e e.g., to “Actor”.
3. 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 one of these properties is selected, the status of the data handler is this particular selected property. For instance, for Actor, we can select the property “`subClassOf`”. At this point, we are still exploring the data source by extending the data handler. Accordingly, the type of the class with this property does not exist yet. Using the dot-operator on this property, the programmer gets a list of classes that are the super-classes of the domain-class. For instance, the class “Artist”. When one class of

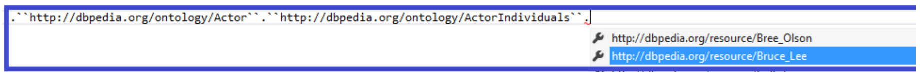
this list is selcted, the status of the data handler is, like in the second case, a certain RDF class (e.g., “Artist”).



```
``http://dbpedia.org/ontology/Actor``.
    http://dbpedia.org/ontology/ActorIndividuals
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
    http://www.w3.org/2000/01/rdf-schema#comment
    http://www.w3.org/2000/01/rdf-schema#isDefinedBy
    http://www.w3.org/2000/01/rdf-schema#label
    http://www.w3.org/2000/01/rdf-schema#subClassOf   type http://www.w3.org/2000/01/rdf-schema#subClassOf =
                                                        nested type http://dbpedia.org/ontology/Artist
```

Fig. 4. Use dot-operator for a class to get properties

– When the collection of individuals is selected, the developer gets a list of individuals. If one is selected, e.g., “Bruce Lee”, the status of the data handler is the current individual, as examplified in Fig. 5.



```
.``http://dbpedia.org/ontology/Actor``.``http://dbpedia.org/ontology/ActorIndividuals``.
                                                  http://dbpedia.org/resource/Bree_Olson
                                                  http://dbpedia.org/resource/Bruce_Lee
```

Fig. 5. Use dot-operator for a class to get individuals

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.

5.2 Type Definitions

If the exploration with the data handler is finished the type of the current state of the data handler is created. An exploration is finished if no further dot operator is applied and instead the programmer presses `return`.

The type bridge has a twofold contribution to the programming environment that is using the bridge. First, it provides the signature from the information space in terms of a name-space, type names, properties and literals. Second, the concrete data and the structure of types are obtained. For instance, if a type for an RDF class is created all the properties of this type are obtained from the data source and accordingly incorporated in the type definition. An essential aspect is that types are created *on-demand*, i.e., lazily, only if the information is required in the host program then the corresponding type is created. Thus, in theory the data source can be very large or even infinite.

Types that are generated by the type bridge are treated as built-in types of the programming language. Accordingly, the usual type operations can be applied too, including type casts and type inference.

6 Scalability Assessment and Implementation

The type bridge is implemented in F#[9]. It can be used as a library (dll-file) within F#. The type bridge has been tested with several data sources like bio2rdf and DBpedia as the largest data source. The implementation is scalable with such large data sources. When exploring the data source using the data handler with the dot-operator, the list of suggestions (as described in Sect. 5.1) is shown to the programmer in less than a second.

The source code, the compiled library and additional descriptions and explanations with illustrating examples can be found at our project Web site[10].

7 Related Work

Related work can be distinguished into two groups: the exploration of data sources (data access) and the integration or mapping of RDF data into programming environments or program code.

The exploration and visualization of data sources allow users without SPARQL experiences an easy access to information from linked data sources. Systems like tFacet [2], Facet [3] and gFacet [4] offer faceted exploration of linked data sources. gFacet provides a graph facet for browsing and fFacet provides a tree view. The visualizing parts of a data source in combination with navigation is studied in [5]. However, in this work the focus is clearly on the visualization part. Compared to our type bridge, none of these approaches consider the integration and typing.

OData[11] is a protocol that aims at querying external data via a RESTful service. OData contains a data model, which is base on an ER model, and accordingly, the protocol describes types and data. Mappings from OData applications to SPARQL endpoints are implemented in the ODATA SPARQL project[12].

Several work studies the mapping between RDF data / RDF triples of program elements. In [6], mapping principles for RDF triples to objects in object-oriented programming languages are presented. This work also includes language extensions for the integration. In [7], an extension of the object-oriented language C# is presented. The approach offers features to order to represent OWL constructs in C#. The typing is similar to our type creation. The specified types are also created at compile time and are integrated into the type system of C#. However, the developer must know the structure (schema) of the ontology in order to define the corresponding types in an OWL-like manner. In contrast, our type bridge can be used for arbitrary RDF data sources due to the exploration support by using the data handler.

Persistence layers offer a kind of access interface from programs to ontologies. Àgogo [8] is a model-driven approach that offers a development environment to

[9] The F# Software Foundation: `http://fsharp.org/`
[10] RDF Type Bridge: `https://github.com/groener/RDFTypeBridge`
[11] OData (Open Data Protocol): `http://www.odata.org/`
[12] OData SPARQL: `https://github.com/BrightstarDB/odata-sparql`

build mappings between program code and OWL ontologies, and to automatically generate an API to access the ontology. OntoMDE [9] is also model driven approach to generate ontology APIs. However, both approaches are suited for accessing and integrating a particular known ontology. Besides this, there is no support of powerful on demand typing and IDE development support as in our RDF type bridge.

Basic mapping principles are applied in code generation, in which RDF data are mapped to programming structures. This is implemented in RDFReactor [10] and Jena [11].

In other application domains, integration and typing is studied in different dimensions. In [12], structural typing, as known from some functional languages, for Java is presented. The principle is comparable to our rules for type definition specifications. The same holds for the specifications for strongly typed heterogeneous collections in [13, 14]. These techniques do not allow for the exploration of properties in data sources, as it is needed for unknown RDF sources.

Tipola [15] is an algorithm and tool for typing DBpedia entities. The types are obtained by natural language definitions from Wikipedia in addition with techniques like word sense disambiguation based on WordNet. This seems feasible for entities where natural languages are available, while our typing approach directly reflects the ontological structure of the RDF data source in the type system.

8 Conclusion

The key investigation of this paper was the integration of RDF data sources into statically typed programs. These statically programs offer great benefits for the developer during the program design, e.g., by interactive type checking, type suggestion and auto-completion. At run-time, statically programs prevent errors and exceptions that are caused by incompatible types. In the realm of data programming, first attempts already studied the integration of well-known schematized data sources into statically typed programs. In this paper, we go one step further and apply similar principles for RDF data sources where schema information is often missing, incomplete or the used vocabulary is unknown to the programmer. To remedy this, we use a data handler to interactively retrieve data from the RDF data source and integrate the retrieved data into the static type system. These types, which are provided by the RDF type bridge, are treated in the programming environment like other built-in types.

As a next step, we plan to extend the type bridge to other more complex representations like the Web ontology language (OWL). For this purpose, a variety additional constructs must be considered in the source exploration, but also in the way how types can be specified.

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. Hildebrand, M., van Ossenbruggen, J., Hardman, L.: /facet: A Browser for Heterogeneous Semantic Web Repositories. In: International Semantic Web Conference. (2006) 272–285
4. 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
5. Dokulil, J., Katreniaková, J.: Navigation in RDF Data. In: 12th International Conference on Information Visualisation, IEEE Computer Society (2008) 26–31
6. Oren, E., Heitmann, B., Decker, S.: ActiveRDF : Embedding Semantic Web Data into Object-oriented Languages. (2008)
7. Paar, A., Vrandecic, D.: Zhi# - OWL Aware Compilation. In: The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II. Volume 6644 of LNCS., Springer (2011) 315–329
8. Parreiras, F.S., Saathoff, C., Walter, T., Franz, T., Staab, S.: 'a gogo: Automatic Generation of Ontology APIs. In: IEEE Int. Conference on Semantic Computing, IEEE Press (2009)
9. Scheglmann, S., Scherp, A., Staab, S.: Declarative representation of programming access to ontologies. In Simperl, E., Cimiano, P., Polleres, A., Corcho, ., Presutti, V., eds.: ESWC. Volume 7295 of LNCS., Springer (2012) 659–673
10. Völkel, M., Sure, Y.: RDFReactor – From Ontologies to Programmatic Data Access. In: Poster Proceedings of the Fourth International Semantic Web Conference. (2005)
11. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: JENA: Implementing the Semantic Web Recommendations. In: 13th International World Wide Web conference on Alternate track papers & posters, ACM (2004) 74–83
12. Gil, J., Maman, I.: Whiteoak: Introducing Structural Typing into Java. In: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA, ACM (2008) 73–90
13. Bracha, G., Lindstrom, G.: Modularity Meets Inheritance. In: Proceedings: 4th International Conference on Computer Languages, IEEE Computer Society Press (1992) 282–290
14. Kiselyov, O., Lämmel, R., Schupke, K.: Strongly Typed Heterogeneous Collections. In: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell. Haskell '04, ACM (2004) 96–107
15. Gangemi, A., Nuzzolese, A.G., Presutti, V., Draicchio, F., Musetti, A., Ciancarini, P.: Automatic Typing of DBpedia Entities. In: International Semantic Web Conference (ISWC). (2012) 65–81