IDE Integrated RDF Exploration, Access and RDF-based Code Typing with LITEQ

Stefan Scheglmann[1], Ralf Lämmel[2], Martin Leinberger[1], Steffen Staab[1], Matthias Thimm[1], Evelyne Viegas[3]

[1]Institute for Web Science and Technologies, University of Koblenz-Landau, Germany
[2] The Software Languages Team, University of Koblenz-Landau, Germany
[3] Microsoft Research Redmond, US

Abstract. In order to access RDF data in Software development, one needs to deal with challenges concerning the integration of one or several RDF data sources into a host programming language. LITEQ allows for exploring an RDF data source and mapping the data schema and the data itself from this RDF data source into the programming environment for easy reuse by the developer. Core to LITEQ is a novel kind of path query language, NPQL, that allows for both extensional queries returning data and intensional queries returning class descriptions. This demo presents a prototype of LITEQ that supports such a type mapping as well as autocompletion for NPQL queries.

1 Introduction

The Resource Description Framework (RDF) is the core technology used in many machine-readable information sources on the Web. RDF has primarily been developed for consumption by applications rather than for direct use by humans. While the flexibility of RDF facilitates the design and publication of data on the Web, it complicates the integration of RDF data sources into applications. For example, it is almost impossible for a developer to know the structure of the data source beforehand. Additionally, there is an impedance mismatch between the way classes or types are used in programming languages compared to how classes are structuring RDF data, cf. [**?**,**?**,**?**,**?**].

To address these challenges, we present LITEQ, a paradigm for querying RDF data, mapping it for use in a host language, and strongly typing it for harvesting the full benefits of advanced compiler technology. It contains mechanisms to map RDF types into code types and allows for embedding query expressions into a host programming language. In addition it is developed with an autocompletion feature in mind so that the autocompletion together with static types can be used to alleviate problems with the unknown structure of RDF data.

In this demo, we present a prototype implementing the language independent LITEQ paradigm for the F# [**?**] programming language. It builds on top of the F# type provider[1] mechanism, which makes the prototype usable in arbitrary IDEs supporting the F# language. In this demo paper, we demonstrate the feasibility and usability of the LITEQ prototype and its integration into an IDE like Visual Studio.

[1] `http://msdn.microsoft.com/en-us/library/hh156509.aspx`

2 Accessing RDF data in Software Development

To illustrate both the challenges encountered by a developer when integrating and reusing RDF data in a programming environment and the contributions of LITEQ, we present some tasks which are prerequisites for working with any data source.

T1 Schema exploration: Initially the structure and the content of data sources are unknown to the developer. In order to identify RDF types that are important for main functionalities of a target application, the developer has to explore the data source and gather information about selected RDF types that he later wants to access in his application.

T2 Code type creation: Once the developer has enough information, he can design and implement his code types and their hierarchy in the host language.

T3 Data querying: Then, the developer uses the schema information in to define queries.

T4 Object creation and manipulation: Given the extensional queries, the developer can retrieve RDF objects and map them into program objects as well as access and manipulate their values.

In the conventional way, the developer could explore the Schema (T1) using a series of SPARQL queries. He can manually write down his own code types based on the RDF types (T2) and formulate SPARQL queries as plain strings in his code (T3). He can then use the results of his written queries to instantiate his previously created classes and actually work with his types (T4). There are several problems with such an approach: Exploring a schema in SPARQL is cumbersome and requires advanced knowledge of SPARQL. Also, creating the types in the code is a recurring task. Lastly, formulating SPARQL queries as plain strings is problematic as errors of all kind will only surface at runtime.

3 Node Path Query Language (NPQL)

Core to LITEQ is a novel path query language, NPQL, that combines type mapping, data querying, and autocompletion to solve the challenges formulated in the previous section. This query language does not stand for itself but is supposed to be embedded into a host language (Fig. 1).

NPQL Syntax Every NPQL expression starts with an URI representing an RDF class. Three different kind of operators allow for the traversal of the RDF schema:

(Op 1) The subtype navigation operator "$\triangledown$" refines the current selected RDF type to one of its direct subclasses.

(OP 2) The property navigation operator "$\triangleright$" expects a property that may be reached from the currently selected RDF type. This property is used as an edge to navigate to the next node, which is defined as the range type of that property.

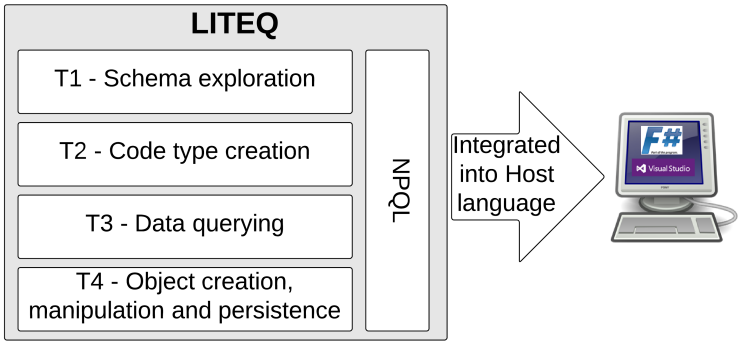

Fig. 1: Integrated into a host language, NPQL can be used to solve the 4 presented tasks.

(OP 3) The property restriction operator "$\triangleleft$" expects a property and uses this property to restrict the extension of the currently selected RDF node. However, it does not traverse the RDF graph further[2].

Using the `FOAF` vocabulary, an example could be the expression 1, which uses a subtype navigation to navigate from `foaf:Agent` to `foaf:Person` and a property navigation via the `foaf:workplaceHomepage` property to the final `foaf:Document` type.

$$\texttt{foaf:Agent} \triangledown \texttt{foaf:Person} \triangleright \texttt{workplaceHomepage} \tag{1}$$

NPQL Semantics: Depending on the context of use, NPQL expressions are evaluated using one or two of the three different semantics:
(Sem 1) The *extensional semantics* of NPQL provides us with an evaluation function, that evaluates an NPQL expression to a set of URIs (the extension). This semantics is used during data querying (T4).
(Sem 2) The *intensional semantics* provides us with an evaluation function, that maps an NPQL expression to an URI. This RDF type URI can be used in order to gather all necessary information, like hierarchy and properties, and to generate the corresponding code type. This semantics is used during code type creation (T2) but also during the retrieval and manipulation of RDF objects (T4).
(Sem 3) The *autocompletion semantics* can complete suggestions for partially written queries. This is possible, because at every step of query writing, we can give a formal semantics of what the intensional meaning of the partially written query is. Using this, an IDE of the host language can provide autocompletion for NPQL expressions. This is used during schema exploration (T1) and query formulation (T3).

4 Usage in F#

To explore the schema of an RDF data source (T1), one first creates a connection to the store. Then, the autocompletion semantics can help the developer to understand his

[2] It is a topic of our future research whether it may be of advantage to dynamically form a description logics like anonymous class expression $ex : Creature \sqcap \exists ex : hasOwner$ and uses this for typing in the host programming language.

data source. Figure 2 depicts such an exploration. The connection is created using the `RDFStore` object. The developer looks at the properties of `foaf:Person`, which is a subtype of `foaf:Agent`.

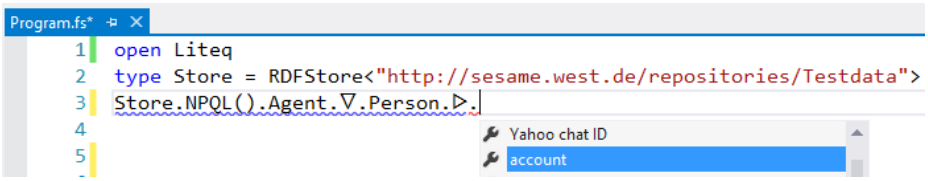

Fig. 2: Build-in autocompletion used for exploring a data source.

The store connection can also be used to access types from the store directly (T2). Again, this type selection is supported by autocompletion. Figure 3 shows expressions that create code types for `foaf:Person` and `foaf:Organization` using the intensional semantics of LITEQ.

```
type Person = Store.Person.Intension
type Organization = Store.Organization.Intension
```

Fig. 3: Intensionally evaluated queries that yield types.

Querying data (T3), like all `foaf:Persons` with a `Skype ID` can also be done using NPQL. Figure 4 shows such a query using the property restriction to restrict the set of all `foaf:Person` typed URIs to those who have a `Skype ID`. By applying an extensional evaluation of the expression, a sequence of person instances is returned. These instances are typed according to the intensional semantics.

```
// Retrieve all persons that have a Skype Id
let personsWithSkypeAccounts = Store.NPQL().Person.◁.``Skype ID``.Extension
```

Fig. 4: Querying for all persons with a Skype ID.

The developer can also use the code types, such as the instances returned in the statement shown in Fig. 4, to modify the data in the store (T4). Returned types, such as shown in Fig. 3 can be used to instantiate new entities. Figure 5 depicts such an instantiation and manipulation of a new `foaf:Person`. Every change made to such an object is automatically propagated to the underlying Triplestore.

5 Implementation

The prototype shown in the screenshots is written in F# and builds on its type provider technology. To enable the NPQL query-object and usable types, the schema of the store is analyzed on IDE startup. Based on this analysis, the type provider can generate the classes that are necessary to integrate NPQL query expressions and intensional types. Intensional and extensional evaluation semantics as well as mappings that convert NPQL expressions to SPARQL queries are encoded in this type provider. Every part of the query is essentially a method, adding a triple pattern to a query. The exten-

```
let newPerson = new Person(newPersonUri)
newPerson.firstName <- [ "Jacob" ]
newPerson.familyName <- [ "Jansson" ]
newPerson.``Skype ID`` <- [ "jacob.jansson" ; "someguy42" ]
```

Fig. 5: Creation and manipulation of a new Person instance.

sional evaluation is then implemented as a SPARQL query that includes all these triple patterns.

Links to the current implementation, a technical report with an extended discussion of the NPQL semantics as well as a screencast of the current LITEQ implementation, showing the autocompletion can be found at `http://west.uni-koblenz.de/Research/systems/liteq`.

6 Conclusion and further work

In this demo paper we presented an implementation of LITEQ for the F# programming language. LITEQ allows for querying, code type creation, and data access of RDF data from within the host language IDE and tries to alleviate the arising challenges. It facilitates a syntax-checked query language, the node path query language (NPQL) to explore, navigate, and query unknown RDF data sources via SPARQL endpoints. The prototypical implementation of LITEQ makes use of the strong type system of F#. Thus, type safety is guaranteed and the generated types are treated as built-in types. Due to the origin of F# in the .NET Framework, types generated by the F# implementation of LITEQ can be used in other .Net languages like C#.

Acknowledgements This work has been supported by Microsoft.

References

1. V. Eisenberg and Y. Kanza. Ruby on semantic web. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *ICDE2011*, pages 1324–1327. IEEE Computer Society, 2011.
2. L. Hart and P. Emery. OWL Full and UML 2.0 Compared. http://uk.builder.com/whitepapers/0and39026692and60093347p-39001028qand00.htm, 2004.
3. A. Kalyanpur, D. J. Pastor, S. Battle, and J. A. Padget. Automatic Mapping of OWL Ontologies into Java. In *SEKE2004*, 2004.
4. T. Rahmani, D. Oberle, and M. Dahms. An adjustable transformation from owl to ecore. In *MoDELS2010*, volume 6395 of *LNCS*, pages 243–257. Springer, 2010.
5. D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamaa, 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 MSR-TR-2012-101, Microsoft Research, 2012.