LITEQ: Language Integrated Types, Extensions and Queries for RDF Graphs

Stefan Scheglmann¹, Gerd Gröner², Ralf Lämmel³, Martin Leinberger¹, Steffen Staab¹, Matthias Thimm¹, Evelyne Viegas⁴

¹Institute for Web Science and Technologies, University of Koblenz-Landau, Germany
²Paluno - Ruhr Institute for Software Technology, University of Duisburg-Essen, Germany
³The Software Languages Team, University of Koblenz-Landau, Germany
⁴Microsoft Research Redmond, US

Abstract. In order to program with RDF data, 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 programmer. 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. While NPQL has limited expressivity compared to SPARQL it has been conceived for supporting interactive data source exploration with an autocompletion semantics for incomplete queries in order to support the programmer at development time.

1 Introduction

The Resource Description Framework (RDF) is currently the core technology used in many machine-readable information sources on the Web. RDF has primarily been developed for the consumption by applications rather than 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. A paradigm that targets to support the access of RDF data from a host programming language must overcome several challenges. First, accessing an external data source requires knowledge about the structure of the data source and its vocabulary. As the core idea of RDF is the serendipitous use of data sources, it is almost impossible for programmers to know the structures before they start developing their programs. Therefore, an approach that simplifies access to RDF sources should include a mechanism for exploring and understanding the RDF data source at development time. Second, 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. [7, 5, 14, 3]. Third, a query and integration language must be easily readable and usable for an incremental exploration of RDF data sources. Finally, when code in a host language describes how RDF data is processed by the resulting program, it is desirable that the RDF data representations are typed, so that type safety can be ensured in order to avoid run time errors and exceptions.

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 taking the full benefits of advanced compiler technology. In particular, LITEQ comprises the following features:
Feature 1: The node path query language (NPQL), a schema and data query language with an intuitive syntax providing operators for the navigation and exploration of RDF graphs, together with an extensional semantics, which clearly defines the retrieval of RDF resources, cf. Sections 4.1 and 4.2.

Feature 2: An intensional semantics for NPQL, which clearly defines the retrieval of RDF schema information, cf. Section 4.3.

Feature 3: Based on the extensional and the intensional semantics, the variable free notation of NPQL supports the definition of an autocompletion semantics. The autocompletion semantics assigns a formal result set to partially written, i.e. incomplete queries. The result set of an incomplete query according to the autocompletion semantics supports the incremental query writing process (cf. Section 4.4).

Feature 4: LITEQ allows for the use of NPQL queries from within a host programming language. This allows for using NPQL to create representations for RDF types and sets of entities in the host language, cf. Section 5.

Additionally, we start with a brief introduction of RDF and our terminology in Section 2. We introduce a running example in Section 3. We present related work in Section 6 and conclude the paper with an outlook on further work in Section 7.

2 Foundations

The Resource Description Framework (RDF) is a data model for representing and annotating data on the Web. RDF data sources consist of RDF graphs, which are sets of RDF statements (triples), as specified in Def. 1.

Definition 1 (RDF Graph). Let $B$ (blank nodes), $L$ (literals), and $U$ (URIs) be disjoint sets. An RDF graph $G$ is a set of RDF triples: $G = \{(s p o) \mid (s p o) \in (B \cup U) \times U \times (B \cup L \cup U)\}$. In each RDF triple, $s$ is referred to as subject, $p$ as predicate and $o$ as object.

We consider a RDF graph, in which each triple (statement) can be derived from an original graph by using RDF entailment. In the further course of this paper, we assume that RDF schema information is complete such that each predicate between a subject and an object is appropriately typed with a domain class that the subject belongs to and a co-domain class that the object belongs to. In addition, we assume the for each property only one single domain class and one single co-domain class exists. While a lack of such structural information can be addressed by schema induction [8], we want to ignore the intricacies of schema induction in this paper and require that all RDF graphs contain complete schema information.

Furthermore, we need to map between the types defined in an RDF graph, i.e. instances of rdfs:Class, and types (or classes or records) as defined in a programming environment. To avoid terminological conflicts, we refer to the former as RDF types and to the latter as code types from here on.

1. RDF Primer: http://www.w3.org/TR/rdf-primer last visit January 13th, 2014
3 Scenario and Requirements

To illustrate both the challenges encountered by a programmer when integrating and reusing a RDF data in a programming environment and the contributions of LITEQ, we introduce the running example that will be used in the remainder of this paper. Then, some basic tasks are introduced that are prerequisites for working with any data source. Finally, we show how the running example could be solved in a naive way, without the help of LITEQ. In Section 5, we will use the naive approach to illustrate the way that LITEQ facilitates a programmer’s tasks.

3.1 Running example

A programmer called Bill has to create an application for municipal administration that allows for managing dogs that live in his city. The application should offer the following functionalities:

1. Managing all registered creatures in the community. This includes browsing, adding, removing and editing all information about dogs from inside of the application.
2. A fee reminder function that reminds all dog owners to pay the dog license fee.

All schema data and all data we consider here are published in the RDF graph $G$ as described in Listing 1 and 2, respectively. $G$ contains the RDF types `ex:Dog` (line 5-7) and `ex:Person` (line 8-11), which are subclasses of `ex:Creature` (line 1-4), and the class `ex:Food` (line 12). There are three properties in the schema: `ex:hasOwner` (line 14-17), `ex:hasName` (lines 18-21) and `ex:eats` (lines 22-25).

Listing 1: The RDF Schema in $G$

```
1  ex:Creature rdf:type ex:Class.
2  rdfs:subClassOf rdfs:Resource.
3  ex:Dog rdf:type ex:Class.
4  rdfs:range ex:Person.
5  ex:Person rdf:type ex:Class.
6  rdfs:range ex:Creature.
7  ex:Food rdf:type ex:Class.
```

Listing 2: The RDF Data in $G$

```
1  ex:hasso rdf:type ex:Dog;  ex:hasOwner ex:tom.
2  ex:bello rdf:type ex:Dog;  ex:hasName "Bello".
3  ex:tom rdf:type ex:Person;  ex:hasName "Tom".
```

3.2 Tasks

Before the programmer can implement his program logic, there are several tasks that needs to be addressed first. These tasks strongly correlate with the four features introduced in Section 1.

T1 Schema exploration: At the beginning, the structure and the content of the data source are completely unknown to the developer. In order to identify RDF types that are of importance for main functionalities of his 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. Of special interest are the rdfs:Class statements and rdfs:subClassOf relationships as well as all rdfs:Property entities with their rdfs:domain and rdfs:range values.

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 Query formulation: The programmer uses the schema information in order to define a query.

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

3.3 Naive approach

In a naive approach, Bill must first explore the data source to understand its content, possibly using a series of SPARQL queries (Task T1). Based on his understanding he may define the code types, including code type hierarchies that mirror the RDF type hierarchies and properties (Task T2), as illustrated in Listing 3.

Listing 3: Code types for Running Example

```java
1 class Creature { 9  class Dog inherit
2   id: Uri 10   Creature {
3   hasName: string 11   hasOwner: Person
4   eats: Food 12   }
5  } 13 }
6 14
7 class Person inherit 15 class Food { }
8  Creature { }
```

For the second functionality, the fee reminder, Bill needs to retrieve representations for all dog owners, entities related from ex:Dog over ex:hasOwner, in his application. He formulates a SPARQL query (Task T3) as indicated in Listing 4, lines 1-3. Creating objects from the result set of the SPARQL query, he may access the data (Task T4; cf. lines 4-6).

Listing 4: Creating the set of all dog owners

```java
1 string queryString = "SELECT ?uri WHERE {
```
Thus, Bill can code the intended application as presented in Section 3.1.

4 The Node Path Query Language (NPQL)

The core idea behind NPQL is that one may use the same NPQL expression in different contexts, but depending on the context an extensional semantics, an intensional semantics or an autocompletion semantics is used to evaluate the expression. First, the extensional semantics is good for data access (Section 4.2). Second, the intensional semantics (Section 4.3) allows for retrieving schema information, which may also be used to type the result of the extensional semantics. Third, the autocompletion semantics builds on extensional and intensional semantics in order to evaluate incomplete NPQL queries for supporting the developer in query writing (cf. Section 4.4). This section starts with the NPQL syntax that lies behind all these three evaluation strategies (Section 4.1).

4.1 NPQL Syntax

In this section, we formalize the NPQL syntax as a BNF in Definition 2 and give a brief introduction into the different NPQL constructs.

**Definition 2 (EBNF Grammar).** The start symbol of the following grammar is `nodepath`, the terminal symbols are all elements of \( U = \{ u_0, \ldots \} \) which itself refers to a valid URI as defined in Definition 1.

\[
\begin{align*}
\text{nodepath} & ::= \text{nodepath nodestep} | \text{URI} \\
\text{nodestep} & ::= >' \text{URI} | .' \text{URI} | <' \text{URI} | [ ' \text{nominalexpr} ' ] \\
\text{nominalexpr} & ::= \text{URI} | '\text{URI} '; \text{nominalexpr} \\
\text{URI} & ::= u_0 | u_1 | u_2 | \ldots
\end{align*}
\]

If \( a, b, c \) are valid URIs then \( a > b < c, a.b[c], a < b.c \) are valid NPQL expressions. Every NPQL expression starts with an URI which should be an RDF class in order to get a meaningful statement. The different operators of NPQL then allow the traversal of the RDF schema from this entry point on. As the most general RDF class is always \texttt{rdf:Resource} this is the most natural starting point for most expressions. The different operators then allow to navigate through the schema by e.g. subclass relationships “\( > \)” or properties “\( . \)”.

We will provide two alternative formal semantics for NPQL in the next two sections. But first, we will illustrate the intuitive meaning behind the individual operators by means of some examples.

**Listing 5: NPQL Query Examples**

1. \texttt{rdf:Resource}\texttt{>ex:Creature}\texttt{>ex:Dog}
2. \texttt{rdf:Resource}\texttt{>ex:Creature}\texttt{>ex:Dog.ex:hasOwner}
3. \texttt{rdf:Resource}\texttt{>ex:Creature}\texttt{>ex:Dog[ex:hasOwner}
4. \texttt{rdf:Resource}\texttt{>ex:Creature}\texttt{>ex:Dog[ex:Hasso]}

(1) The subtype navigation operator “\(>\)” refines the current selected RDF type to one of its direct subclasses. The expression in Listing 5 line 1, will refine the selected starting point `rdf:Resource`, first to `ex:Creature` and then to `ex:Dog`.

(2) The property navigation operator “\(\cdot\)” 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. So extending the NPQL expression from the Listing 5 line 2, moves the selected RDF type from `ex:Dog` (the class of its domain of `ex:hasOwner`) to `ex:Person` the class of its co-domain.

(3) The property restriction operator “\(<\)” expects a property and using this property to restrict the extension of the currently selected RDF node. However, it does not traverse the RDF graph further. To illustrate this, let us assume a property restriction choosing `ex:hasOwner`, cf. Listing 5 line 3. This will not change the currently selected RDF type but restrict its extension to all URIs of RDF type `ex:Dog` for which there is also an `ex:hasOwner` relation.

(4) The nominal operator “[ ]” expects a list of instances of the currently selected RDF node. Traversal may continue with the RDF node appearing immediately before the nominal operator. By using the nominal operator Listing 5 line 4, we are able to restrict the extension of the RDF type `ex:Dog` (the set of all dogs) to the set that contains only `ex:Hasso`.

In the next sections we provide two alternative semantics for NPQL, an extensional semantics that interprets NPQL expressions as sets of URIs (the extension) and an intensional semantics that interprets NPQL expressions as types (the intension).

### 4.2 NPQL Extensional Semantics

The extensional semantics of NPQL given in Definition 3 provides us with an evaluation function \([ \cdot ]_G^e\) for NPQL expressions. This function evaluates an NPQL expression to a set of URIs (the extension).

**Definition 3 (Extension of a Node Path).** The evaluation function \([ \cdot ]_G^e\) for the extensional semantics of NPQL with respect to an RDF graph \(G\) is a function that maps an NPQL expression to a set of URIs, i.e., \([ \cdot ]_G^e: \text{nodepath} \mapsto \{u_1, \ldots, u_n\}\), and is recursively defined as follows.

\[
\begin{align*}
[u]_G^e &= \{u' \mid (u' \text{ rdf: type } u) \in G\} \\
[nodepath > u]_G^e &= [nodepath]_G^e \cap [u]_G^e \\
[nodepath.u]_G^e &= \{u \mid \exists u', u'': (u \text{ rdfs: range } u') \in G, (u' \text{ rdf: type } u') \in G, u'' \in [nodepath]_G^e : (u'' u u) \in G\} \\
[nodepath < u]_G^e &= [nodepath]_G^e \cap \{u' \mid \exists u'': (u'' u u') \in G\} \\
[nodepath[u_1, \ldots, u_n]]_G^e &= \{u_1, \ldots, u_n\} \cap [nodepath]_G^e
\end{align*}
\]

Listing 6 illustrates a step-wise evaluation of Listing 5 line 3:

\footnote{It is an open research question whether it may be of advantage to dynamically form a description logics like anonymous class expression `ex : Creature \(\sqcap\exists ex : hasOwner\)` and use this for typing in the host programming language. This will be a topic of our future research.}
Listing 6: NPQL Extensional Evaluation

1. \[ [\text{rdf:Resource}]^G_G = \{ u \mid (u \text{ rdf:type rdf:Resource}) \in G \} \]
2. \[ [\text{rdf:Resource}>ex:Creature]^G_G = \{ \text{ex:Hasso, ex:Bello} \} \]
3. \[ [\text{rdf:Resource}>ex:Creature>ex:Dog]^G_G = \{ \text{ex:Hasso, ex:Bello} \} \]
4. \[ [\text{rdf:Resource}>ex:Creature>ex:Dog<ex:hasOwner}]^G_G = \{ \text{ex:Hasso} \} \]

In Lines 3 and 4 you can see very clearly how the use of the property restriction operator “<” restricts the intensional evaluation. The expression in line 3 evaluates to all instances of \text{ex:Dog} in the graph, while the expression in line 4 is restricted to \text{ex:Dog} instances for which a \text{ex:hasOwner} exists.

4.3 NPQL Intensional Semantics

The intensional semantics given in Definition 3 provides us with an evaluation function \([\cdot]^G \), that maps an NPQL expressions to a URI. This RDF type URI can be used in order to gather all necessary information in order to generate the corresponding code type. Like the set of all property URIs this RDF type is domain to or the necessary hierarchical relationships.

Definition 4 (Intension of a Node Path). The evaluation function \([\cdot]^G \), for the intensional semantics of NPQL with respect to an RDF graph \( G \) is a function that maps an NPQL expression to an URI, i.e., \( [\cdot]^G : \text{nodepath} \mapsto u \) with \( u \in U \), and is recursively defined as follows.

- \( [u]^G = u \)
- \( [\text{nodepath} > u]^G = u \)
- \( [\text{nodepath}.u]^G = \hat{u} \) where \((u \text{ rdfs:range} \hat{u}) \in G \)
- \( [\text{nodepath} < u]^G = [\text{nodepath}]^G \)
- \( [\text{nodepath}[u_1, \ldots, u_n]]^G = [\text{nodepath}]^G \)

Listing 7 illustrates a step-wise evaluation of the intensional semantics of Listing 5 line 2: Line 3 and 4 demonstrate nicely how the use of the type navigation operator “.” changes the result of the intensional evaluation from \text{ex:Dog} to the co-domain of the property \text{hasOwner}, \text{ex:Person}.

Listing 7: NPQL Intensional Evaluation

1. \[ [\text{rdf:Resource}]^G = \text{rdf:Resource} \]
2. \[ [\text{rdf:Resource}>ex:Creature]^G = \text{ex:Creature} \]
3. \[ [\text{rdf:Resource}>ex:Creature>ex:Dog]^G = \text{ex:Dog} \]
4. \[ [\text{rdf:Resource}>ex:Creature>ex:Dog<ex:hasOwner}]^G = \text{ex:Person} \]

4.4 Autocompletion Semantics

NPQL queries can be embedded in a host programming language and can be written directly in the IDE. NPQL supports autocompletion, because at every step of query writing, we can give a formal semantics of what the intensional meaning of the partially written query is. The evaluation functions given in Definition 5 explicitly refer to
incomplete queries according to the grammar in Definition 2. We refer to this meaning of a partial query by autocompletion functions as follows:

**Definition 5 (Autocompletion Evaluation Functions).** The evaluation function \( \langle \cdot \rangle^G_a \) for the autocompletion of NPQL expression with respect to an RDF graph \( G \) maps an NPQL expression and an operator to a set of URIs, i.e., \( \langle \cdot \rangle^G_a : \text{nodepath} \mapsto \{ u_1, \ldots, u_n \} \).

\[
\langle \text{nodepath} \rangle^G_a \! = \! \{ u | (u \text{rdfs:subClassOf} [\text{nodepath}]^G_i) \in G \land \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (u' \text{rdfs:subClassOf} [\text{nodepath}]^G_i) \in G \} \\
\langle \text{nodepath}. \rangle^G_a \! = \! \{ u | (u \text{rdfs:domain} [\text{nodepath}]^G_i) \in G \cup \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (u \text{rdfs:domain} u') \in G \} \\
\langle \text{nodepath} \rangle^G_a \! = \! \{ u | (u \text{rdfs:domain} [\text{nodepath}]^G_i) \in G \cup \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (u \text{rdfs:domain} u') \in G \} \\
\langle \text{nodepath} \rangle^G_a \! = \! \{ u | (u \text{rdfs:domain} [\text{nodepath}]^G_i) \in G \cup \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (u \text{rdfs:domain} u') \in G \} \\
\langle \text{nodepath} \rangle^G_a \! = \! \{ u | (u \text{rdf:type} [\text{nodepath}]^G_i) \in G \} \\
\langle \text{nodepath}[u_0, \ldots, u_i] \rangle^G_a \! = \! \{ u | (u \text{rdf:type} [\text{nodepath}]^G_i) \in G, \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad u \neq u_j \text{ where } j = 0, 1, 2, \ldots \} \\
\]

Listing 8 illustrates step-wise the autocompletion suggestions while writing the query from Listing 5 line 3: As Bill does not know anything yet about the graph, so he starts at the general rdf:Resource and decides to use subtype navigation “\( \rangle \)” For “rdf:Resource” autocompletion suggests all direct subtypes, cf. Definition 5 rule one and Listing 8 line 1. Bill chooses ex:Creature from the suggested and decides again to perform a subtype navigation. All direct subtypes of ex:Creature are suggested, cf. line 2. Finally, after choosing ex:Dog and property restriction “\( \langle \rangle \)”, this evaluates to all properties of ex:Dog and its superclasses, cf. rule three and listing line 3. A screencast of the current LITEQ implementation, showing the autocompletion can be found at [http://west.uni-koblenz.de/Research/systems/liteq](http://west.uni-koblenz.de/Research/systems/liteq).

**Listing 8: NPQL Extensional Evaluation**

1. \( \langle \text{rdf:Resource} \rangle^G_a \! = \! \{ \text{ex:Creature, ex:Food} \} \) \\
2. \( \langle \text{rdf:Resource}\text{ex:Creature} \rangle^G_a \! = \! \{ \text{ex:Dog, ex:Person} \} \) \\
3. \( \langle \text{rdf:Resource}\text{ex:Creature}\text{ex:Dog} \rangle^G_a \! = \! \\
\quad \{ \text{ex:hasOwner, ex:hasName, ex:eats} \} \)

**5 LITEQ in the Host Language**

In this section, we show how the four programmer’s tasks described in Section 3.2, are facilitated by LITEQ while the programmer is developing code in the IDE.

**5.1 Exploring Schema and Creating Code Types**

Bill will explore the data source using the autocompletion of his IDE. This will help him to explore the schema and choose the RDF types he intends to use (Task 1). The types
are chosen by writing LITEQ queries that are then intensionally evaluated. Queries that
generate the Dog and Person classes are shown in Listing 9\footnote{It is safe to assume that in most languages, a data access object will be used to write the NPQL queries. Therefore, we included such an object named graph in this code listing.}. The definitions \texttt{class Dog} and \texttt{class Person} should be seen as type aliases for the types that will actually be generated.

Listing 9: Queries that create types

```
class Dog =
  graph.Intension(rdf:Resource>ex:Creature>ex:Dog)
class Person =
  graph.Intension(rdf:Resource>ex:Creature>ex:Person)
```

The intensional evaluation will derive a URI of a type and automatically invoke
Algorithm 1, which generates the code types including the type hierarchy (Task 2).

**Algorithm 1**: Type creation

```
name : generateType
input : u, the URI of the RDF type
TCache, the type cache
output: Type, The code type for the given RDF type description

begin
  /* Check if RDF type already in cache */
  if u ∈ TCache.keys() then
    return TCache.get(u);
  /* Otherwise create a new type */
  t := createTypeDef(u);
  TCache.add(u, t);
  /* Collecting and adding super types */
  foreach uSuper ∈ \{u | ∃uSuper : (u rdfs:subClassOf uSuper) ∈ G\} do
    tSuper := generateType(uSuper);
    t.addSuperType(tSuper);
  /* Collecting and adding properties */
  foreach uProp ∈ \{u | ∃uProp : (uProp rdfs:domain u) ∈ G\} do
    uRange := (u | (uProp rdfs:range u));
    tRange := generateType(uRange);
    t.addProperty(uRange, tRange);
return t;
```

During the actual type creation for every of these NPQL expressions, a meta pro-
gramming method building these code types is called (see Algorithm 1). However, it is
not enough to just build the type defined by the NPQL expression, in order to preserve
the RDF type hierarchy in the code types, it is necessary to also create code types for
the related super RDF types, Algorithm 1 lines 6-8. In order to add typed properties to
the code type, we have to check for the co-domain RDF type for all related properties
and create these as well, lines 9-12. Due to the fact that multiple NPQL expressions
can point to the same RDF type and in order to create a code type hierarchy, the type creation has to be aware of the already created types. This can be solved by the use of caching mechanism for the generated types, lines 2-5. The resulting class definitions look similar to the ones shown in Listing 3.

5.2 Retrieving and Accessing the data

The remaining part for Bill is to implement his program logic. Just like in Section 3.3, he needs the set of all dog owners in order to implement the tax reminder functionality (Task 3). He uses the extensional evaluation of NPQL queries to access the necessary data (Task 4). Listing 10 shows the code he might use to get all dog owners and print their names for his dog fee reminder functionality.

Listing 10: Querying for dog owners

```java
Set<Person> dogowners = graph.Extension(rdf:Resource>
  ex:Creature>ex:Dog.hasOwner)
for o in dogowners
  print o.hasName
```

In order to instantiate concrete objects, the target class must be determined by evaluating the query intensionally and then created. In the listing above, this would set the class to Person, as the co-domain of ex:hasOwner is Person. Then the query can be evaluated extensionally and the created class can be used to instantiate the host language object for every RDF individual.

6 Related Work

LITEQ is generally related to three different research directions - query languages for RDF, the integration of data access into a host languages in particular mappings of RDF into the object oriented paradigm and exploration tools for unknown RDF data sources.

Considering query languages, a number of different languages are available for RDF. In general, we can distinguish two different ways of querying graph data as RDF. First querying as a graph matching problem, matching subgraphs descriptions against the data, like in SPARQL queries. Second by using a graph traversal language, like Gremlin or Cypher or the languages mentioned in [17]. Examples of graph traversal languages for RDF data are nSPARQL [13], a language with focus on navigation through RDF data, or GuLP [4], which can include preferential attachment into its queries. However, there are two major differences between these two, exemplary chosen, languages and LITEQ. While nSPARQL and GuLP both use their own evaluation implementations, LITEQ exploits the widely spread SPARQL support by mapping its queries to SPARQL. The second difference lies in LITEQs type generation. Through that, a IDE can provide native autocompletion for the many parts of the queries - a feature that is

---

3 Gremlin graph traversal language https://github.com/tinkerpop/gremlin/wiki last visit January 13th, 2014
3 Cypher graph traversal language in Neo4J http://docs.neo4j.org/chunked/stable/cypher-query-lang.html last visit January 13th, 2014
rare among RDF query languages. Additionally, no switch in context is necessary when
exploring new data sources. SPARQL queries can best be tested in a dedicated web in-
terface, while LITEQ queries are best used inside an IDE, which will later be used to
write the program logic.

The problem of accessing and integrating RDF data in programming environments
has already been recognized as a challenge in various work. Most approaches focus on
ontology driven code generation in order to realize RDF access in the programming
environment. Frameworks like ActiveRDF [9], AliBaba\(^4\), OWL2Java [7], Jastor\(^5\), RD-
FReactor\(^6\), OntologyBeanGenerator\(^7\), and Ágogo [12] were developed in the past. An
overview can be found at Tripresso\(^8\), a project web site on mapping RDF to the object-
oriented world. The common goal for all these frameworks is to translate the concepts
of the ontology into an object-oriented representation.

All the aforementioned approaches rely on external exploration of the data, dedi-
cated type declarations, and a code generation steps in order to provided the desired
data representations in the host language. In contrast to that, the compile time meta-
programming features of F\#, type providers\(^9\) integrate all these steps directly into the
host language. Type providers support the integration of information sources into F\# [16]
such that an external data sources is directly available during development time. Sev-
eral Type Providers demonstrate the integration of large data sources on the Web, e.g.
the Freebase Type Provider that allows for the navigation within the graph-structure of
Freebase\(^10\). Our implementation of LITEQ for F\# uses the type provider principles.
However the main difference to a naive use of type providers in the context of RDF is
that LITEQ extends the explorative mechanism of type providers by the NPQL query
language.

While the previous examples are targeted at specific languages, some concepts
which aim to transcend language exist. Ágogo [12] and OntoMDE [15] are programming-
language independent model driven approaches for automatically generating ontology
APIs. They introduce intermediate steps in order to captures domain concepts necessary
to map ontologies to object-oriented representations.

The basic mapping principles of RDF triples to objects common to the previously
presented approaches [10] and programming language extensions to integrate RDF or
OWL constructs [11] have already been explored. LITEQ also uses these principles.
However, there are two main differences that sets LITEQ apart. It can provide type def-
initions when editing code while the necessary subset of types at compilation time. [11]
also presents a language extension for C\# that offers features to represent OWL con-
structs in C\# and that is able to create the types at compile time. In contrast to LITEQ,
there is no means for querying and navigating unknown data sources. The developer must be aware of the structure of the ontology.

Other research work has a dedicated focus on exploration and visualization of Web data sources. The main motivation of this work is to allow users without SPARQL experiences an easy means to get information from RDF data sources. tFacet [1] and gFacet [6] are tools for faceted exploration of RDF data sources via SPARQL endpoints. tFacet 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 [2], but with the focus on visualization aspects like optimization of the displayed graph area. In contrast to LITEQ, 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.

7 Conclusion and further work

This paper has presented LITEQ, a new programming paradigm to access and integrate representations for RDF data into typed programming languages. LITEQ facilitates an expressive query language, the node path query language (NPQL) to explore and navigate external unknown RDF data sources via SPARQL endpoints. NPQL supports incremental query writing in terms of variable-free node path expressions. Based on the node path, which is the result of a navigation through the RDF graph, LITEQ offers intensional and extension evaluation of the node path, retrieving either schema information in terms of classes and properties or RDF resources. Both schema information as well as RDF resources can be used in the programming language to define types at development time and to instantiate these types at run time. 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.

For further work, we plan to extend LITEQ by several features. First, we plan to define conditional type casts such that casting between types, which are built according to external RDF data sources, depend on conditions, e.g., on the presence of certain property statements. A similar principle can be applied to iterators, i.e., the iteration of a collection depends on characteristics of the collection’s members that must be satisfied. Second, the execution of functions and methods can depend on contracts. For instance, arguments of function, which have a certain type, need to satisfy conditions (i.e., the contract) in order to allow the execution of a function or method. Third, we will extend the current extensional description of node path towards the OWL-like set semantics in order to allow for more complex type specifications.

Acknowledgements This work has been supported by Microsoft.

References