Semantic Web Application development with LITEQ

Martin Leinberger[1], Stefan Scheglmann[1], Ralf Lämmel[2], 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. The Semantic Web is intended as a web of machine readable data where every data source can be the data provider for different kinds of applications. However, due to a lack of support it is still cumbersome to work with RDF data in modern, object-oriented programming languages, in particular if the data source is only available through a SPARQL endpoint without further documentation or published schema information. In this setting, it is desirable to have an integrated tool-chain that helps to understand the data source during development and supports the developer in the creation of persistent data objects. To tackle these issues, we introduce LITEQ, a paradigm for integrating RDF data sources into programming languages and strongly typing the data. Additionally, we report on two use cases and show that compared to existing approaches LITEQ performs competitively according to the Halstead metric.

1 Introduction

RDF has primarily been developed for consumption by applications rather than direct use by humans. However, its design choices, which facilitate the design and publication of data on the Web, complicate the integration of RDF data into applications as its principles do not tie in smoothly with modern concepts in programming languages.

A paradigm that aims to support a developer in integrating RDF data into his application must overcome several challenges. First, accessing an external data source requires knowledge about the structure of the data source and its vocabulary.

Therefore, an approach that provides integration of RDF data sources should include a mechanism for exploring and understanding the RDF data source at development time. Second, it is desirable that an exploration and integration mechanism is well-integrated, easily learnable and useable. The integration should be either on programming language level or at least in the used IDE (*Integrated Development Environment*), as this allows for seamless exploration and integration during application development. Third, there is an impedance mismatch between the way classes or types are used in programming languages compared to how classes are used in RDF data, cf. [8, 14, 6, 3]. For this reason, an approach for integrating RDF data into a programming language must define a clear mapping between these two mismatching paradigms.

To address these challenges, we present LITEQ, a paradigm for querying RDF data, mapping it for use in a host language, and strongly typing[1] it for taking the full benefits

[1] By "strongly typed", we refer to languages where it is not possible to have unchecked runtime type errors, e.g., through validating the program by static type checking before execution.

of advanced compiler technology. The core of LITEQ is the Node Path Query Language (NPQL), a variable-free schema and data query language, which comprises the following features:

1. various operators for the navigation and exploration of RDF graphs (Section 4.1),
2. an intensional semantics, which defines the retrieval of RDF schema information and enables our implementation of LITEQ to provide persistent code types of RDF entities (Section 4.3),
3. an extensional semantics, which defines the retrieval of RDF resources and enables our implementation of LITEQ to construct persistent objects from the retrieved data (Section 4.3), and
4. an autocompletion semantics, which assigns a formal result set to partially written, i.e. incomplete, queries; this allows an incremental query writing process (Section 4.2).

The fully functional prototype of LITEQ (called $\mathsf{LITEQ}^{F\#}$) is written in F#, a member of Microsoft's .NET family. Up until now, there has been only limited support for RDF in the .NET framework. Our approach $\mathsf{LITEQ}^{F\#}$ is currently being prepared to be added to the FSharp.Data project, a library containing access mechanisms for many different structured data formats. In this in-use paper, we report also on two use cases that show the applicability of LITEQ for practical problems. The prototype is documented at our website[2] and can be downloaded from github[3].

The remainder of the paper is organized as follows. In Section 2, we start with a brief introduction of RDF and F# followed by a general process overview of how a data model based on the Semantic Web is implemented in Section 3. We then describe LITEQs features in Section 4 and its implementation in Section 5 before we show the feasibility of our approach based on two different use cases in Section 6. This is followed by a discussion of related work in Section 7 and a short summary in Section 8.

2 Foundations

The Resource Description Framework[4] is a data model for representing data on the Web. An RDF a data source consist of a graph, which is a set of RDF statements (triples).

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: $G = \{(s\ p\ o)|(s\ p\ o) \in (B \cup U) \times U \times (B \cup L \cup U)\ \}$. In each RDF triple, s is referred to as subject, p as predicate and o as object.*

[2] LITEQ Project at WeST `http://west.uni-koblenz.de/Research/systems/liteq`, last visit 12th Mai, 2014

[3] LITEQ on Github `https://github.com/Institute-Web-Science-and-Technologies/Liteq`, last visit 12th Mai, 2014

[4] RDF Primer: `http://www.w3.org/TR/rdf-primer` last visit January 13th, 2014

In the further course of this paper, we assume that such a graph is enriched with complete RDF schema information 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 for each property only one single domain class and one single co-domain class exists. If such strict assumptions are not met by the RDF data sources, LITEQ provides configuration possibilities that can make up for not meeting this assumption and which will be explained in Section 4.

The F# language is a statically typed, object-oriented and functional-first programming language based on Microsoft's .NET framework. It has been fully open sourced by the F# Software Foundation and features a cross-platform compiler that allows F# to run on Windows, Linux and OS X. Libraries written in F# can be used in all .NET languages.

3 Process overview for implementing a use case

Generally, the process of developing an application using an RDF data source can be described by five different tasks that must be addressed during development. First, the developer must design an initial data model based on the requirements of his application (Task 1). He explores the schema of his data set and identifies the RDF classes and properties that are of interest to him (Task 2). To this end, he either has external documentation that he refers to, or has to explore the schema using a series of SPARQL queries. In this step, he will also align his previously created data model to the data source. He can then leave the design phase and enter the coding phase, where he implements the data model in the programming language (Task 3). He can continue with designing the queries he needs to access the data (Task 4). Lastly, as such queries in programming languages usually return generic result sets, he can map the results of his previously written queries onto the code types he created (Task 5). Figure 1 summarizes the process.

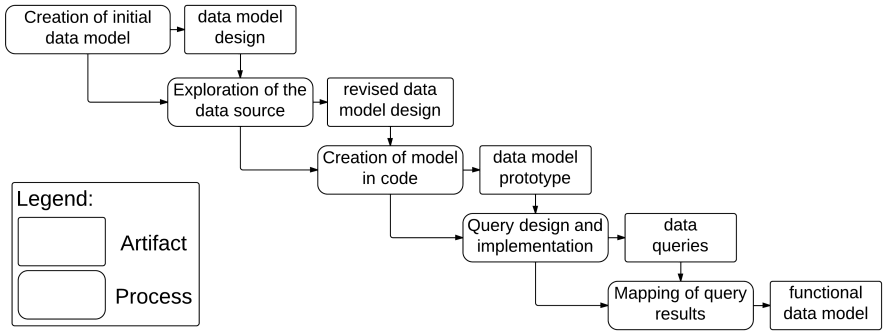

Fig. 1: Process Overview for creating a data model based on a RDF data set.

4 Using $LITEQ^{F\#}$ in practice

The current implementation of LITEQ in F# is IDE-independent and can be used in

many F#-IDE like Microsoft's Visual Studio[5], Xamarin Studio[6], or Monodevelop[7].

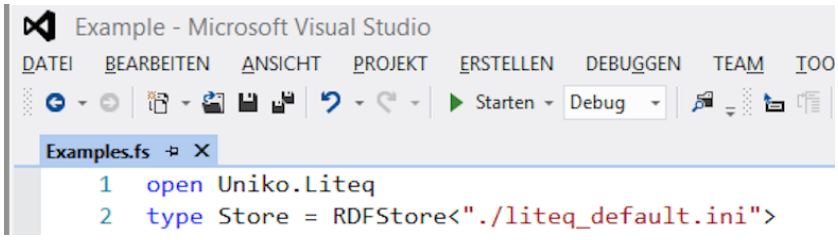

Fig. 2: Setting up LITEQ

As with any other library, LITEQ must be referenced and imported before it can be used. This is shown in Figure 2. The developer imports the LITEQ API, cf. line 1, and sets up a data access object using a specific configuration. Listing 3 shows an example of such a configuration file.

```
serverUri=http://.../openrdf-sesame/repositories/Jamendo
isReadOnly=false
prefixfile=prefixes.ini
```

Listing 3: A LITEQ configuration File.

The first line of this configuration file defines a SPARQL endpoint that is acting as a data source. The key isReadOnly defines whether the endpoint also accepts SPARQL update queries or not[8]. In the example configuration, it is set to false, meaning that the store can be updated. The prefixfile property points to a file in which RDF vocabulary prefixes are defined. This is optional, as the namespaces and prefixes may not be known beforehand, but improves the readability of expressions as the shortened versions can be used instead of the full URI. As many real world data sources separate the schema from the data and provide the schema only as a separate file, the optional property schemaFile has been introduced. This property can be used to include such an external schema file. If the property is not given, the schema is assumed to be queryable via the given SPARQL endpoint.

The data access object returned by the initialization can then be used to perform the different operations provided by LITEQ, such as exploration, navigation, and data retrieval.

4.1 Node Path Query Language

Core to LITEQ is the Node Path Query Language (NPQL), a schema and data navigation language which supports the developer in navigating and exploring the RDF data source from within his programming environment.

[5] http://www.visualstudio.com/ last visit April 29th, 2014
[6] https://xamarin.com/studio last visit April 29th, 2014
[7] http://monodevelop.com/ last visit April 29th, 2014
[8] Objects constructed by LITEQ$^{F\#}$ can automatically update the store when assigned new data.

The NPQL method of the data access object mentioned above allows for navigating the schema using NPQL expressions, cf. Figure 4.

```
6  Store.NPQL().``foaf:Agent``
7  Store.NPQL().``foaf:Agent``.v.``mo:MusicArtist``
8  Store.NPQL().``foaf:Agent``.v.``mo:MusicArtist``
9      .``->``.``mo:member_of``.``foaf:Group``
10 Store.NPQL().``foaf:Agent``.v.``mo:MusicArtist``
11     .``->``.``mo:member_of``.``foaf:Group``.``<-``.``foaf:skypeID``
```

Fig. 4: Navigating the Schema using NPQL

Every NPQL expression starts with an URI of the target graph. In case of the example shown in Figure 4, line 10, we start with `foaf:Agent` as entry point for our NPQL exploration. The different operators of NPQL then allow for traversal of the RDF schema from this entry point on. We provide three different operators in NPQL, which allow for navigating through the schema in different ways.

1. The subtype navigation operator "v" refines the currently selected RDF type to one of its direct subclasses. The expression in Figure 4 line 7, will refine the selected starting point `foaf:Agent` to `mo:MusicArtist`.
2. The property navigation operator "`->`" 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 Figure 4 lines 8–9, shifts the selection from `foaf:Agent` to the property and further to its `foaf:Group`, its range.
3. The property restriction operator "`<-`" expects a property and uses this property to restrict the extension of the currently selected RDF node. To illustrate this, let us assume a property restriction choosing `foaf:skypeID`, cf. Figure 4 line 10–11. This will not change the currently selected RDF type but restrict its extension to all URIs of RDF type `foaf:Group` for which there is also an `foaf:skypeID` relation.

Using these three operators, the developer can explore the data. Furthermore, he can use the very same expressions to construct data types and objects from the data and schema, as we will demonstrate in Section 4.3.

4.2 Autocompletion Support

LITEQ$^{F\#}$ provides an autocompletion mechanism for NPQL, i. e., at every step of query writing we can formally define the meaning of the partially written query and provide suggestions for completion. This was done in order to support the developer during the exploration of the data source as described in Task 2, Section 3. Figure 5 shows the autocompletion feature when writing an NPQL query in Visual Studio. The developer starts with `mo:MusicArtist` and decides to perform a property navigation. This evaluates to the list of all properties, that have `mo:MusicArtist` or one of its supertypes is its domain.

In a further step, the developer reduces the results by defining the starting letters of the properties to be "ma", cf. Section 4.1. As shown in Fig. 6, this reduces the results to `foaf:made` and `foaf:maker`.

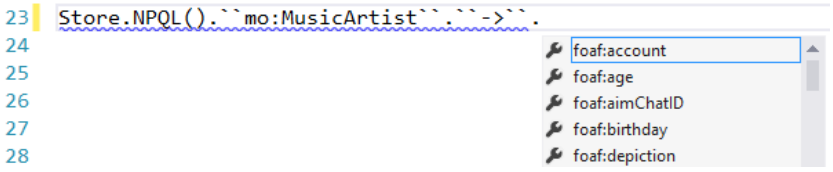

Fig. 5: LITEQs Autocompletion support.

Fig. 6: Refinement of autocompletion suggestions.

4.3 Evaluation of NPQL expressions

In order to interpret NPQL expressions, LITEQ provides two different functions, (Extension and Intension) that evaluate complete NPQL expressions based on two formal semantics. We only provide an informal overview here, the formal intensional and extensional semantics of NPQL can be found in our technical report published on `https://west.uni-koblenz.de/Research/systems/liteq`

Intension The intensional semantics evaluates an NPQL expression to a code type (the intension). This relates to Task 3 (data model creation) as presented in Section 3. Figure 7 shows the intension of `mo:MusicArtist`, cf. line 11. In the figure, its code type MusicArtist is subsequently used to instantiate a new artist newArtist, cf. line 12. The declared MusicArtist class has property definitions of all properties as they were defined in the RDF schema, e. g., the `foaf:name` property of the new instance which is set to myBandName, cf. line 13.

```
11  type MusicArtist = Store.``mo:MusicArtist``.Intension
12  let newArtist = new MusicArtist("http://.../artist/1234")
13  newArtist.``foaf:name`` <- [ "myBandName" ]
```

Fig. 7: Declaring code types using LITEQ.

Extension The extensional semantics evaluates an NPQL expression to its set of RDF objects. The extension could either be a set of URIs (in case of RDF types) or to a set of domain/range tuples (in case of properties). This relates to Task 4 (Query design) as presented in Section 3. However, LITEQ will also automatically type the result of such an extensional evaluation, returning instances of code types as if they were generated through an intensional evaluation. Therefore, extensional evaluation also relates to Task 5 (Mapping of query results). Figure 8 shows such a extensionally evaluated NPQL expression, cf. line 15. This statement returns a sequence of all `mo:MusicArtist` to allArtists. This sequence is subsequently iterated in order to print the music records of all artists, cf. lines 16-17.

```
17  let allArtists = Store.NPQL().``mo:MusicArtist``.Extension
18 for artist in allArtists do
19     printfn "Artist %A made the following records: %A" artist artist.``foaf:made``
```

Fig. 8: Querying for all artists and iterating through the result.

In $LITEQ^{F\#}$, both, the intensional and extensional evaluation is implemented by transforming the expressions into SPARQL queries which are then executed against a SPARQL endpoint.

5 Implementation of LITEQ$^{F\#}$

$LITEQ^{F\#}$ is based on the type provider[9] framework. This framework allows us to generate code types based on the available schema information.

Figure 9 shows a class diagram of the current implementation. It is centered around the LITEQ type provider, which serves as an entry point. It is also responsible for building the navigational classes for NPQL. All code necessary for the mapping from RDF types to actual programming language types is contained in the TypeMapper class. Both need access to the triples and schema information, which can currently either be a generic SPARQL endpoint via the SPARQL HTTP interface or a dump of the schema.

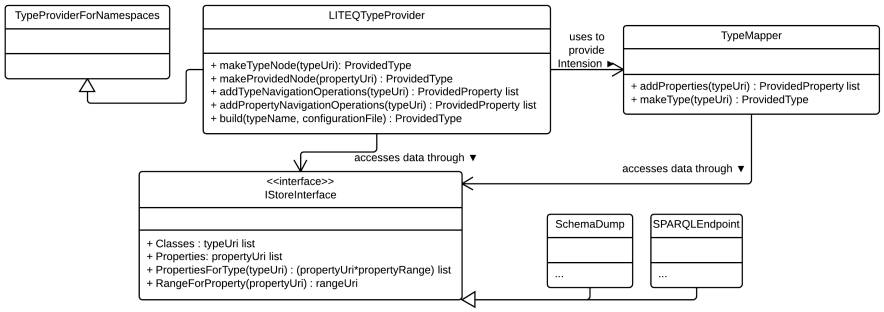

Fig. 9: Simplified (for brevity) class diagram of the implementation.

Figure 10 shows the runtime behavior of the system. At some point, the library is called via the *build* method. This will trigger the creation of all necessary classes for NPQL queries and usage in the language. However, they newly created classes do not yet contain any properties as this turned out to be to slow in practice. Properties are only added once the IDE asks the object for its properties. The objects contain callbacks to the methods of the LITEQ type provider or type mapper that will return all properties for the specific object. This step finalizes the object.

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

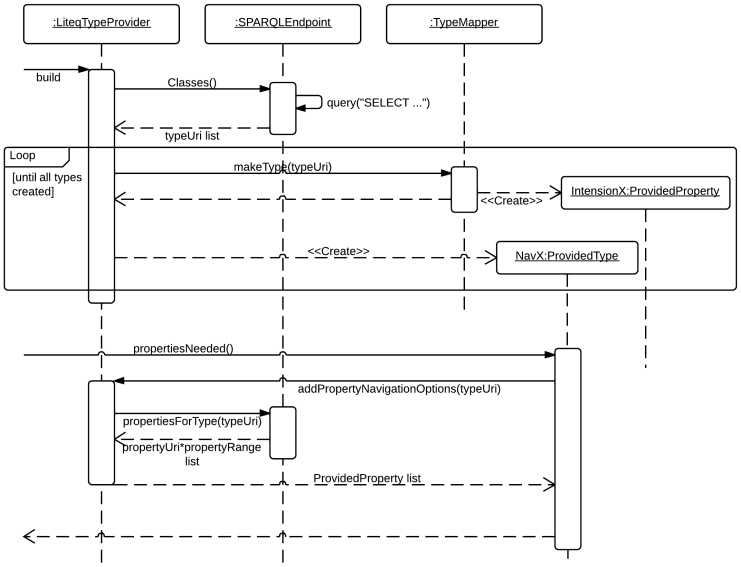

Fig. 10: Sequence diagram of behavior during usage.

6 Use Cases

In order to show the feasibility of LITEQ$^{F\#}$, we have chosen to implement two tasks using a traditional framework such as dotNetRDF and the LITEQ$^{F\#}$ approach. We then compare the implementations using the Halstead metric [5] to determine the difficulty and effort of the implementations.

6.1 Use Case: Creating a new artist object and listing all tracks

An RDF data source in a programming language should be easily queryable but also modifiable. Therefore, our first simple use case is about inserting new data and iterating over a subset:

R1 The program shall create a new instance of type music artist and assign a name to that instance.
R2 The program shall iterate over all instances of type music artist and print the music records associated with them.

The data that serves as the input for this task comes from the Jamendo data set[10].

6.1.1 Implementation using the dotNetRDF framework A SPARQL implementation of this task relies on the SPARQL Update functionality. This type of SPARQL

[10] `http://moustaki.org/resources/jamendo-rdf.tar.gz`, last accessed on 08.05.14

queries allow the insertion of new triples into the triple store. Listing 12 shows how the code for inserting a new artist with a specific `foaf:name` looks like. SPARQL queries are written as plain strings with specific symbols marking substrings that are to be replaced with concrete values, such as the music artist URI or the name, right before the update is executed. To simplify the SPARQL query, namespaces are often bound in a separate step outside of the actual query.

```
let connection = new SesameHttpProtocolConnector(
                                "http://.../openrdf-sesame",
                                "Jamendo")

// Defining Update query
let query = new SparqlParameterizedString("INSERT DATA {
    @instanceUri a mo:MusicArtist .
    @instanceUri foaf:name @name . }")
query.Namespaces.AddNamespace("mo",
    new Uri("http://purl.org/ontology/mo/"))
query.Namespaces.AddNamespace("foaf",
    new Uri("http://xmlns.com/foaf/0.1/"))

// Setting specific values and executing
query.SetUri("instanceUri",
    new Uri("http://artist/1234"))
query.SetLiteral("name", "myBandName")
connection.Update( query.ToString() )
```

Listing 11: Inserting a new MusicArtist with a specific name.

Iterating over the music artists is similar. Listing 12 shows how it can be implemented. Again, the query is defined as a string with the namespaces being bound separately. In this specific query, the expected result contains always the artist URI and the record URI that was made by this artist. The result of this query is initially a result set, which can be grouped by artists by piping[11] it to the *group by* function and specifying a projection function. The computation will result in tuples containing the specific artist and a list of result sets that contain the query results about the artist. These result sets associated with the different artists can then be mapped to a list containing only the record URIs. The resulting tuple containing artist and the list of records they made can then be printed to the console.

```
let query' = new SparqlParameterizedString("SELECT
    ?artist ?record WHERE {
    ?artist a mo:MusicArtist .
    ?artist foaf:made ?record .
}")
query'.Namespaces.AddNamespace("foaf",
    new Uri("http://xmlns.com/foaf/0.1/"))
query'.Namespaces.AddNamespace("mo",
```

[11] The pipe operator f |> g is used to pass the result of one computation to the next one. An equal statement has the form g(f()).

```
    new Uri("http://purl.org/ontology/mo/"))

let results =
    connection.Query(query'.ToString()):?>SparqlResultSet
    |> Seq.groupBy( fun res ->
        res.Value("artist").ToString() )
    |> Seq.map( fun (artist, results) ->
        artist, results |> Seq.map(fun res ->
            res.Value("record").ToString())
    )
for (artist, records) in results do
    printfn "Artist %A made the following records: %A"
```

Listing 12: Iterating over artists and printing records.

6.1.2 Implementation using LITEQ The LITEQ implementation has already been used as an example. Creating a new music artist is an intensional evaluation of the `mo:MusicArtist` URI. The resulting type can afterwards be instantiated and assigned data via setter methods as displayed in Fig. 7.

Iterating over the music artists in the data source is an extensional evaluation of `mo:MusicArtist`—this returns a sequence of music artists that is iterable as done in Fig. 8.

6.1.3 Comparison In order to evaluate the two different approaches, we chose to apply the Halstead metric. According to Halstead, a program can be seen as a collection of tokens that are classified as operators and operands. Halstead then defines:

Definition 2 (Halsteads complexity measure). *Let n_1 be the number of distinct operators, N_1 the total number of operators, n_2 the number of distinct operands, and N_2 the total number of operands. Then one has:*

Program vocabulary $n = n_1 + n_2$
Program length $N = N_1 + N_2$
Volume $V = N \times log_2 n$
Difficulty $D = \frac{n_1}{2} \times \frac{N_2}{n_2}$
Effort $E = D \times V$
Necessary time $T = \frac{E}{18} seconds$

Of special interest to us is the difficulty of a program, which expresses how hard it is to understand it a code review, and the theoretical time that is needed to code such a program. When applying the metric, we defined that all language constructs, such as "let .. =", "type .. =" or "for .. in .. do" and access operators were to be counted as operations. The same holds for static parameters[12] and "," creating tuples in mappings[13].

[12] <> indicate a static parameter

[13] We omitted them when they were used as separators for method parameters due to language differences between F#, for which LITEQ is optimized and C#, which dotNetRDF was written in.

The remaining type and method names were counted as operands. The same holds for strings, except for those representing SPARQL queries. SPARQL queries were counted as one operator for the general construct (INSERT DATA .. or SELECT .. WHERE ..), one operation to form a specific triple pattern and three operands per triple. In LITEQ namespaces are defined in an separate file, so we did not count the code necessary to add namespace definitions, as shown in Listing 12, By applying this metric to the Use Case, we get the results shown in Table 2[14].

	dotNetRDF	**LITEQ**
No. of distinct operators (n_1)	12	7
No. of distinct operands (n_2)	39	18
Total no. of operators (N_1)	36	15
Total no. of operands (N_2)	63	25
Program Vocabulary	51	25
Program length	99	40
Program Volume	561,57	185,75
Difficulty	9,69	4,86
Effort	5442,91	902,97
Time needed	302s	50s

Table 1: Halsteads complexity applied to Use Case 1.

6.2 Use Case: Analyzing number of EU seats for countries

RDF data sets, especially governmental data, are often used for analysis and visualization. Therefore, the second use case visualizes the number of EU seats hold by different countries:

R1 The program shall select all countries that hold at least one EU seat
R2 The program shall then transform this data into a suitable structure and visualize it

The data set used for this task was a dump of the DBpedia data set[15].

6.2.1 Implementation using the dotNetRDF framework Using the dotNetRDF framework, a developer needs to open the connection to the SPARQL endpoint and can then write a query receiving the name of the country and its number of seats. He then needs to map the result from the result set to a tuple containing a string representing the name of the country and the number of EU seats. The visualization is then only a function call. Listing 13 shows the the necessary code to do so in F#.

[14] The full source code with annotations that describe what we counted as operand and operator or omitted(e.g. brackets) can be found under `http://www.uni-koblenz-landau.de/campus-koblenz/fb4/west/Research/systems/liteq`

[15] Available under `http://wiki.dbpedia.org/Downloads39`, last accessed on 06.05.14

```
let connection = new SparqlRemoteEndpoint(
        new Uri("http://dbpedia.west.uni-koblenz.de:8890/sparql"))
let data =
    connection.QueryWithResultSet("""SELECT ?countryName
    ?numberOfSeats WHERE {
        ?country <http://dbpedia.org/property/euseats>
            ?numberOfSeats .
        ?country <http://xmlns.com/foaf/0.1/name> ?countryName .
        }""")
    |> Seq.map ( fun resultSet ->
        resultSet.Value("countryName").ToString(),
        int(resultSet.Value("numberOfSeats").ToString()
        .Replace("^^http://www.w3.org/2001/XMLSchema#integer",
            ""))
    )

data
|> Chart.Pie
```

Listing 13: Calculating percentage of EU seats.

6.2.2 Implementation using LITEQ All countries holding an EU seat can be selected using an NPQL expression. The expressions returns a sequence of `Country` objects as if created through the intension, the name and number of EU seats can be accessed as in an object model. While they return proper types (strings for the name and ints for the number of EU seats), they return lists of these types as the schema did not specify the cardinality. In order to visualize the countries are mapped to a sequence of tuples containing name and number of EU seats by accessing the corresponding members and taking the first element out of the result list. Listing 14 shows the necessary code to do so.

```
type Store = RDFStore<".\liteq_default.ini">

let euCountries =
    Store.NPQL().``dbpedia:Country``.``<-``.``dbpediapr:euseats``
    .Extension

euCountries
|> Seq.map( fun country ->
    country.``foaf:name``.[0],
        country.``dbpediapr:euseats``.[0] )
|> Chart.Pie
```

Listing 14: Calculating percentage of EU seats using SPARQL.

However, as the DBPedia ontology did not specify any range or domain for the number of EU seats a country holds, we had to extend the schema, in the local schema file, with the specific values to enable this implementation.

6.2.3 Implementation using a custom object model Apart from comparing the implementations in LITEQ and plain SPARQL, we also want to compare to an custom implementation. Such an implementation bases on a object model which incorporates schematic information that exceeds the information currently available in RDFS such as information about cardinalities of properties.

As such object models usually do not incorporate query languages, all countries have to be selected in this scenario. The resulting sequence can then be filtered to contain only countries that hold at least one EU seat. To visualize the data, the same approach as with LITEQ can be used — mapping the sequence of countries to a sequence of tuples containing name and number of seats for the country and passing this to the visualization function. Listing 15 displays such code.

```
connectToStore "http://.../openrdf-sesame/Jamendo"

let euCountries =
    Country.findAllInstances
    |> Seq.filter( fun country -> country.EuSeats > 0 )

euCountries
|> Seq.map( fun country -> country.Name, country.EuSeats )
|> Chart.Pie
```

Listing 15: Calculating percentage of EU seats with an object model.

6.2.4 Comparison To evaluate the three different approaches, we again apply the Halstead metric as defined in Def. 2.

	dotNetRDF	**LITEQ**	**Perfect Object Model**
No. of distinct operators (n_1)	9	7	6
No. of distinct operands (n_2)	25	16	13
Total no. of operators (N_1)	19	18	13
Total no. of operands (N_2)	34	22	20
Program Vocabulary	34	23	19
Program length	53	40	33
Program Volume	269,64	180,94	140,20
Difficulty	6,12	4,81	4,62
Effort	1650,17	870,79	646,99
Time needed	92s	49s	36s

Table 2: Halsteads complexity applied to Use Case 2.

Again, the version using dotNetRDF and SPARQL is the most difficult to understand and slowest to code. LITEQ improves on this while a custom object model can improve on LITEQ.

6.3 Evaluation of the results

The Halstead metric supports our assumption—in both use cases, it is much easier to understand the code implementing such a scenario with LITEQ than using SPARQL

queries. The same holds for the time necessary to write the code, which is, in all cases less for LITEQ than a implementation using dotNetRDF.

When comparing LITEQ to a perfect object model, it can be seen that there is still room for improvement. A better mapping from RDF types to code types manifests itself in less difficulty and less effort. However, if such a better mapping exists, we can incorporate it into LITEQ and get similar results.

7 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. The first one considers querying as a graph matching problem, matching subgraphs descriptions against the data, like in SPARQL queries. The second way is by using a graph traversal language, like Gremlin or Cypher or the languages mentioned in [16]. 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 the type generation provided by LITEQ.

NPQL queries and Description Logics (DL) expressions share some similarities. A DL expressions always describes a concept, its place in the concept lattice and the extension of the concept. Similarly, NPQL expressions can be evaluated extensionally to a set of entities or intensionally to a type description including its place in the type hierarchy. The intensional evaluation of NPQL expressions also consists information about the attributes of the type, in contrast to DL concepts which only contain information about constrains over attributes.

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[16], OWL2Java [8], Jastor[17], RDFReactor[18], OntologyBeanGenerator[19], and Àgogo [12] were developed in the past.

[15] Gremlin graph traversal language `https://github.com/tinkerpop/gremlin/wiki` last visit January 13th, 2014

[15] Cypher graph traversal language in Neo4J `http://docs.neo4j.org/chunked/stable/cypher-query-lang.html` last visit January 13th, 2014

[16] `http://www.openrdf.org/alibaba.jsp` last visit January 13th, 2014

[17] `http://jastor.sourceforge.net/` last visit January 13th, 2014

[18] `http://semanticweb.org/wiki/RDFReactor` last visit January 13th, 2014

[19] `http://protege.cim3.net/cgi-bin/wiki.pl?OntologyBeanGenerator` last visit January 13th, 2014

An overview can be found at Tripresso[20], 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. While the previous examples are targeted at specific languages, some concepts which are language-agnostic language exist. Àgogo [12] and OntoMDE [15] are programming-language independent model driven approaches for automatically generating ontology APIs. They introduce intermediate steps in order to capture domain concepts necessary to map ontologies to object-oriented representations. All the aforementioned approaches rely on external exploration of the data, dedicated type declarations, and code generation steps in order to provide the desired data representations in a programming language.

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. For one, LITEQ has build-in type generation support, that can automatically generate such types. Another examples that also features this is [11]. This is a language extension for C# that offers features to represent OWL constructs in C# and that is able to create the types at compile time. In contrast to LITEQ however, there is no means for querying and navigating unknown data sources. The developer must be fully aware of the structure of the ontology ahead of development time.

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 [7] are tools for faceted exploration of RDF data sources via SPARQL endpoints. fFacet provides a tree view for navigation, while gFacet has a graph view 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.

8 Summary

This paper presented the fully functional prototype of LITEQ$^{F\#}$ targeted for release as part of FSharp.Data library. The implementation features a new paradigm to access and integrate representations for RDF data into typed programming languages. We also showed the feasibility of our approach. The documentation of the current LITEQ$^{F\#}$ library, including a video showing LITEQ in use, can be found on our website[21], while the source code is available at Github[22]. In the near future, we plan to improve performance and stability of the system, before shifting focus and ensuring a smooth integration into

[20] `http://semanticweb.org/wiki/Tripresso` last visit January 13th, 2014

[21] `http://west.uni-koblenz.de/Research/systems/liteq`, last visit 12th Mai, 2014

[22] `https://github.com/Institute-Web-Science-and-Technologies/Liteq`, last visit 12th Mai, 2014

FSharp.Data.

Acknowledgments This work has been supported by Microsoft.

References

1. S. Brunk and P. Heim. tFacet: Hierarchical Faceted Exploration of Semantic Data Using Well-Known Interaction Concepts. In *DCI2011*, volume 817 of *CEUR-WS.org*, pages 31–36, 2011.
2. J. Dokulil and J. Katreniaková. Navigation in RDF Data. In *iV2008*, pages 26–31. IEEE Computer Society, 2008.
3. 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.
4. V. Fionda and G. Pirrò. Querying graphs with preferences. In *CIKM2013*, pages 929–938, 2013.
5. Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.
6. L. Hart and P. Emery. OWL Full and UML 2.0 Compared. http://uk.builder.com/whitepapers/0and39026692and60093347p-39001028qand00.htm, 2004.
7. P. Heim, J. Ziegler, and S. Lohmann. gFacet: A Browser for the Web of Data. In *IMC-SSW2008*, volume 417 of *CEUR-WS*, pages 49–58, 2008.
8. A. Kalyanpur, D. J. Pastor, S. Battle, and J. A. Padget. Automatic Mapping of OWL Ontologies into Java. In *SEKE2004*, 2004.
9. E. Oren, R. Delbru, S. Gerke, A. Haller, and S. Decker. Activerdf: object-oriented semantic web programming. In *WWW2007*, pages 817–824. ACM, 2007.
10. E. Oren, B. Heitmann, and S. Decker. ActiveRDF : Embedding Semantic Web Data into Object-oriented Languages. *J. Web Sem.*, pages 191–202, 2008.
11. A. Paar and Vrandecic D. Zhi# - OWL Aware Compilation. In *ESWC2011*, volume 6644 of *LNCS*, pages 315–329. Springer, 2011.
12. F. S. Parreiras, C. Saathoff, T. Walter, T. Franz, and S. Staab. 'a gogo: Automatic Generation of Ontology APIs. In *ICSC2009*. IEEE Press, 2009.
13. J. Pérez, M. Arenas, and C. Gutierrez. nsparql: A navigational language for rdf. *J. Web Sem.*, 8(4):255–270, 2010.
14. 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.
15. S. Scheglmann, A. Scherp, and S. Staab. Declarative representation of programming access to ontologies. In *ESWC2012*, volume 7295 of *LNCS*, pages 659–673. Springer, 2012.
16. P. T. Wood. Query languages for graph databases. *SIGMOD Record 2012*, 41(1):50–60, 2012.