# Property-based typing with LITEQ

## Programming access to weakly-typed RDF data

Stefan Scheglmann[1], Martin Leinberger[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.** Coding against the semantic web can be quite difficult as the basic concepts of RDF data and programming languages differ greatly. Existing mappings from RDF to programming languages are mostly schema-centric. However, this can be problematic as many data sources lack schematic information. To alleviate this problem, we present a data centric approach that focuses on the properties of the instance data found in RDF and that lets a developer create types in his programming language by specifying properties that need to be present. This resembles a type definition rooted in description logics. We show how such a type definition can look like and demonstrate how a program using such type definitions can can be written.

## 1 Introduction

Access to RDF data from within programs is difficult to realize since, (i) RDF follows a flexible and extensible data model, (ii) schema is often missing or incomplete, and (iii) data RDF type information is missing. In order to establish a robust access from a program to RDF data, a developer faces several challenges. Most of the data sources are defined externally and the developer has only a brief idea of what to find in this data source, therefore he has to explore the data first. Once explored, he has to deal with the impedance mismatch between how RDF types are structuring RDF data, compared to how code types are used in programming languages, [4, 2, 6, 1]. In response to these challenges, we have proposed LITEQ [5], a system that allows for the mapping of RDF schema information into programming language types.

However, using LITEQ in practice has shown that purely relying on RDF schema information for the mapping raises new issues. To alleviate these problems, we have implemented an property-based approach and include it into LITEQ as an alternative way of usage. Using this, a developer is able to create code types by listing properties that should be present in instances of this code type.

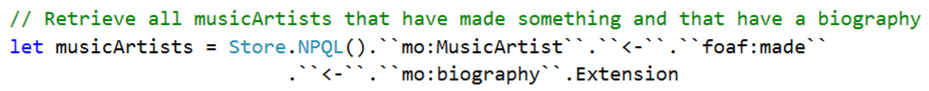
In this demo, we present an implementation of this approach[1] in F#. It supports developers in defining new code types by specifying their properties. This is aided by auto-completion mechanism, which computes its suggestions directly on the instance data without any need of a schema. The approach is intended as an extension to the LITEQ library also presented at the InUse-Track of ISWC 2014.

[1] `http://west.uni-koblenz.de/Research/systems/liteq`

## 2 The LITEQ approach

Typically, the integration of RDF data in a programming environment is a multi-step process. (1) The structure and content of the data source has to be explored, (2) the code types and their hierarchy has to be designed and implemented, then (3) the queries for the concrete data have to be defined, and finally (4) the data can be retrieved and mapped to the predefined code types.

**The NPQL Approach:** LITEQ provides an IDE integrated workflow to cope with all of these tasks. It implements NPQL, a novel path query language, to explore the data source, to define types based on schematic information and to query the data. Returned results of these queries are automatically typed in the programming language. All of this is aided by the autocompletion of the IDE. Figure 1 shows a typical LITEQ expression using an NPQL query in order to retrieve all `mo:MusicArtist` entities which actually have the `foaf:made` and `mo:biography` property defined. The result is returned as a set of objects of the created code type for `mo:MusicArtist`.



```
// Retrieve all musicArtists that have made something and that have a biography
let musicArtists = Store.NPQL().``mo:MusicArtist``.``<-``.``foaf:made``
                               .``<-``.``mo:biography``.Extension
```

Fig. 1: Querying for all music artists that made at least one record and have a biography.

**Property-based Type access with LITEQ:** The example shown above has several problems. Additionally, schema-centric technique only provides code types for which a RDF type is defined in the schema. It is not possible to introduce more specific code types, e.g. if it is known that all entities of interest will have at least one `foaf:made` and `mo:biography` relation, one may like to reflect that by the returned code type. Lastly and most importantly, for all schema-centric access methods, like LITEQ, a more or less complete schema must be present, which is not always the case, especially in Linked Data. To cope with these problems of schema-centric approaches, we introduce the idea of a different data access/typing approach, property-based RDF access in a program [7].

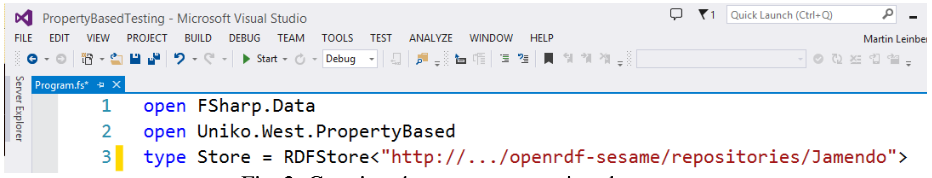
**Property-based Type declaration:** The basic idea is very simple: (1) A code type is defined by a set of properties. (2) Extensionally, a code type represents a anonymous RDF type (or view on the data) which refers to the set of all entities which actually have all the properties defined in the code type. (3) Intensionally, the code type signature is given by the set of properties (for these it provides direct access). All other properties of an instance of this type can only be accessed indirectly in a program using the predicate name. Specifically, this means that a developer might define a type by just declaring a set of properties as the types signature ($Sig$), e.g. $Sig = \{\texttt{foaf:made}, \texttt{mo:biography}\}$. Our approach allows for two different ways to map such a property-based type to a corresponding code type. A flat-mapping which just maps to a `rdf:Resource` representation and only makes the properties explicitly accessible which are defined in the code type signature. Such an unnamed code type refers to the set of all entities sharing the properties in the code types signature ($Sig$), cf. 1.

$$\text{unnamedType}_{Sig} \equiv \exists\texttt{foaf:made} \sqcap \exists\texttt{mo:biography} \tag{1}$$

The second option is named-mapping. Which allows to map the previously unnamed type to a given RDF type. This ensures separation if entities of distinguished types share properties, e.g. music artists and producers given the properties `foaf:made` and `mo:biography`. And to search for other properties stating that their domain is the specified type in order to provide a richer API. The named type is defined as the intersection of the unnamed type for the provided signature ($Sig$) and all entities of the given RDF type (mo:MusicArtist), cf. 2.

$$\text{namedType}_{(Sig,\texttt{mo:MusicArtist})} \equiv \text{unnamedType}_{Sig} \sqcap \texttt{mo:MusicArtist} \qquad (2)$$

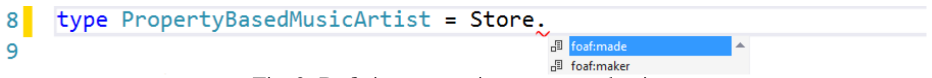
**Type-declaration in Practice** In the following, we will give a brief overview of the new method of access RDF data in a programming environment:
(1) To use the library, the DLL must be referenced from the project. This allows opening the library namespace and creating a store object by passing a URL to a SPARQL endpoint, cf. Figure 2.


PropertyBasedTesting - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP
Quick Launch (Ctrl+Q)
Martin Leinbe
Start
Debug
Program.fs*
Server Explorer

```
open FSharp.Data
open Uniko.West.PropertyBased
type Store = RDFStore<"http://.../openrdf-sesame/repositories/Jamendo">
```

Fig. 2: Creating the type representing the store.

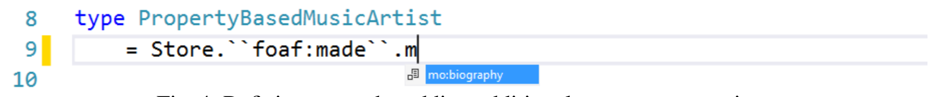
(2) All properties contained in the SPARQL endpoint can be accessed from the store object (cf. Fig. 3).



```
type PropertyBasedMusicArtist = Store.
```

Fig. 3: Defining types via property selection.

(3) Once a property has been chosen, the set of properties presented in the next step is restricted to those properties which have been observed in combination of the previously chosen properties, cf. Figure 4. Here `foaf:made` has already been chosen and the only co-occurring property `mo:biography` is presented for the next selection.

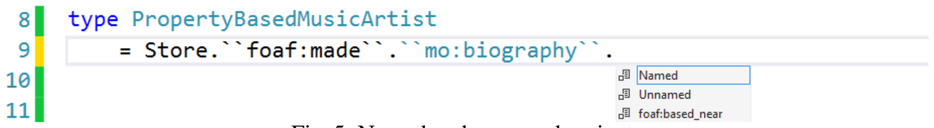


```
type PropertyBasedMusicArtist
    = Store.``foaf:made``.m
```

Fig. 4: Refining a type by adding additional property constraints.

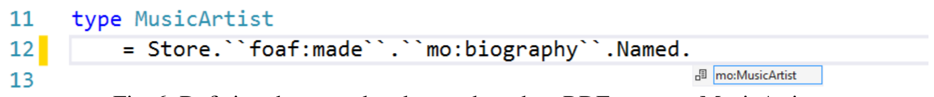
(4) Finally, the developer has to decide on how he likes the new type to be mapped. As mentioned in the previous section, he can choose for an unnamed representation or a named one, cf. Figure 5.
(5) If the developer decides for the named representation, he has to choose the RDF type, cf. Figure 6. In this case, only `mo:MusicArtist` instances contain the specified properties and therefore he can only chose this one RDF type.

The presented method of relying on instance information instead of the schema has a severe drawback when it comes to specifying return code types of properties in the programming language. As one cannot use schematic information, the only chance is



```
type PropertyBasedMusicArtist
    = Store.``foaf:made``.``mo:biography``.
```

Fig. 5: Named and unnamed variants



```
type MusicArtist
    = Store.``foaf:made``.``mo:biography``.Named.
```

Fig. 6: Defining the named code type based on RDF type mo:MusicArtist.

to probe the instance set. However, for big instance sets, this process takes to long to be useful. The current prototype avoids this problem by only probing whether a property returns another RDF resource or a literal and types returned values accordingly.

## 3 Conclusion and Further Work

In this extended abstract, we presented a new way to work with RDF in a programming language - by defining types based on their properties. As an extension of LITEQ, this demo will focus on the new feature of property based type definition, as described in [8, 7] and discussed in [3].

**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. S. Homoceanu, P. Wille, and W. T. Balke. Proswip: Property-based data access for semantic web interactive programming. In *12th International Semantic Web Conference, ISWC 2013*, Sydney, Australia, 2013.
4. A. Kalyanpur, D. J. Pastor, S. Battle, and J. A. Padget. Automatic Mapping of OWL Ontologies into Java. In *SEKE2004*, 2004.
5. M. Leinberger, S. Scheglmann, R. Lämmel, S. Staab, M. Thimm, and E. Viegas. Semantic web application development with liteq. In *International Semantic Web Conference*, 2014.
6. 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.
7. S. Scheglmann and G. Gröner. Property-based Typing for RDF Data. In *PSW 2012, First Workshop on Programming the Semantic Web, Boston, Massachusetts, November 11th, 2012*, 2012.
8. S. Scheglmann, G. Gröner, S. Staab, and R. Lämmel. Incompleteness-aware programming with rdf data. In Evelyne Viegas, Karin Breitman, and Judith Bishop, editors, *DDFP*, pages 11–14. ACM, 2013.
9. 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.