Journal of Multidisciplinary Engineering Science and Technology (JMEST)
ISSN: 3159-0040
Vol. 2 Issue 5, May - 2015

Typed Language for Consuming Linked Data

Li Vladimir
Faculty of Information Technology
Kazakh-British Technical University
Almaty, Kazakhstan

Abstract—Shortly after the release of SPARQL 1.1, Andy Seaborne, the coeditor of the SPARQL recommendation, said: the next step is to not necessarily assume that the access language for RDF is SPARQL, but may be a language that is more navigational in style — there is scope and need for both. In this paper we propose a domain specific scripting language for consuming Linked Data with a simple but appropriate type system, that mixes static and dynamic type checking. The language is designed to be used by Web developers who would like to build applications which consume Linked Data.

Keywords—linked data, triple store, abstract syntax, type system, type checking, type inference

1. INTRODUCTION

In this paper we design a domain-specific typed scripting language for consuming linked data.

2. APPROACH FOR CONSUMING LINKED DATA

There are numerous RDF datasets on the net and any data owner may publish his data as Linked Data by making URI's dereferenceable[1]. Data consumers may want to consume data from number of those sources and combine it into one dataset using links. In that case the application for data consumers will look like Fig 1[3].

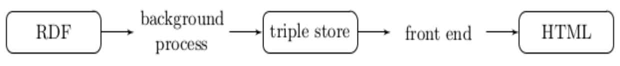

Fig. 1.

In that case there is a local triple store for storing RDF data, which is populated and kept updated through background process which crawls RDF datasets on the net. It should particularly be able to dereference those RDF URIs and update the triple store respectively. The front end is any web application which is built on queries to that triple store.

1. Dereferencing URIs

Back end should be able to dereference URIs to obtain data from RDF sources.It includes 4 steps as described in [1]

1) The client performs a HTTP GET request on a URI identifying a real world object or abstract concept. If the client is a Linked Data application and would prefer an RDF/XML representation of the resource, it sends an Accept:application/rdf+xml header along with the request. HTML browsers would send an Accept: text/html header instead.

2) The server recognizes that the URI identifies a real-world object or abstract concept. As the server cannot return a representation of this resource, it answers using the HTTP 303 See Other response code and sends the client the URI of a Web document that describes the real-world object or abstract concept in the requested format.

3) The client now performs an HTTP GET request on this URI returned by the server.

```
GET /data/Kazakhstan.n3
HTTP/ 1.1
Host: dbpedia.org
Accept: text/n3
```

4) The server answers with a HTTP response code 200 OK and sends the client the requested document, describing the original resource in the requested format.

```
HTTP/ 1.1 200 OK
Date: Tue, 27 Jan 2015 12:35:36 GMT
Content−Type: text/n3
```

2. Higher Level Approach

At a higher level of abstraction we hide dereferencing process and also bind variables to other URIs using pattern matching. Consider following example:[3]

```
from ”dbpedia:Kazakhstan”
select $x
where
(”dbpedia:Kazakhstan” ”dbproperty:capital” $x)
from $x
```

Here from keyword means getting data about dbpedia:Kazakhstan. First the script checks if data is present in local triple store and if not tries to dereference the URI. If successful it loads it into the triple store. select variable where triple binds the value to variable matching the pattern of triples. If matched successfully it tries to dereference new URI with another from construction(if not present in local triple store already). [3]

3. Simple Type Checking

Let's now consider the following example:

```
from dbpedia:Kazakhstan
select $c:string
where
("dbpedia:Kazakhstan" "dbproperty:capital" $c )
( $c > 0 )
```

This would result in type error before running the program because variable $c which is supposed to be of type string is compared to integer 0.

Consider another example:

```
from dbpedia:Kazakhstan
select $a:string
where ("dbpedia:Kazakhstan" "dbontology:areaTotal" $a)
```

This won't result in error during the static type check, but will fail later when we fetch information about "dbontology:areaTotal" and find out it has a type "http://www.w3.org/2001/XMLSchema#double" which matches to double in our grammar.

3. Syntax

1. The Syntax of Types

A type is either a simple datatype or it is a property that allows a simple datatype as its range[3] variables begin with a dollar sign:

```
datatype ::= u r i
| string
| double
| integer
| dateTime
type ::= datatype
| range ( datatype )
var : : = $a
| $b
| $c
| …
```

Datatypes include uris, integers, doubles, strings, dateTime. Typing also applies to filtering triples from rdf. For example, if an URI with a property type is used in the property position of a triple, then the object of that triple can only take the value indicated by the property type.

2. Syntax of Terms

Terms are used to construct RDF triples. Terms can be URIs, variables or values of a simple datatype.

```
term ::= var | uri | string | integer | double | dateTime
```

3. Syntax of Scripts

Scripts are either from select where or from where constructions where multiple variables can be selected based on multiple conditions(queries) or from constructions which dereference and load data to local triple (if not there already).[3]

```
script ::= from term
//Dereference URI
//( and load it into the local triple store if needed )
| from term where c o n d i t i o n s
// Dereference URI
// and assign variables based on conditions
| from term select vartypes where conditions
//Same as previous
// but with variable−type constraints ( $x : integer )
vartype
| var ':' datatype
condition ::= ( term term term ) // pattern for triple matching
| ( boolean ) // boolean constraints for variables
boolean ::= boolean | | boolean
| boolean && boolean
| ! boolean
| term = term
| term < term
…
```

Booleans here are analogues of filters in SPARQL. They take a single argument. The expression can be complex, as long as it returns a boolean value[2]. We use them to compare expressions of the same type.

4. Examples

Selecting data about the capital of Kazakhstan(Astana).[3]

```
from dbpedia : Kazakhstan
where
( dbpedia : Kazakhstan dbpropcapital $x )
from $x
```

Selecting data about people born in Astana

```
from " dbpedia : Kazakhstan "
select
$c : uri
where
("dbpedia:Kazakhstan" "dbprop:capital" $c)
from $c
```

```
select $p
where
($p ”dbprop:birthPlace” $c)
from $p
```

4. Conclusion

Concept of Linked Data enables to crawl the web merging data between datasets. Existing programming environments require many low level details in development. Thus, here we introduce a domain specific high level language for consuming Linked Data where basic syntactic checks(static and dynamic) can be performed.[3]

References

[1] Tom Heath and Christian Bizer (2011) Linked Data: Evolving the Web into a Global Data Space (1st edition). Synthesis Lectures on the Semantic Web: Theory and Technology, 1:1, 1-136. Morgan & Claypool.

[2] Bob DuCharme (2011) Learning SPARQL Querying and Updating with SPARQL 1.1 O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

[3] Gabriel Ciobanu & Ross Horne & Vladimiro Sassone (2011): Local Type Checking for Linked Data Consumers