Local Type Checking for Linked Data Consumers

Gabriel Ciobanu Ross Horne
Romanian Academy, Institute of Computer Science, Blvd. Carol I, no. 8, 700505 Iaşi, Romania
gabriel@info.uaic.ro ross.horne@gmail.com

Vladimiro Sassone
University of Southampton, Electronics and Computer Science, Southampton, United Kingdom
vs@ecs.soton.ac.uk

The Web of Linked Data is the cumulation of over a decade of work by the Web standards community in their effort to make data more Web-like. We provide an introduction to the Web of Linked Data from the perspective of a Web developer that would like to build an application using Linked Data. We identify a weakness in the development stack as being a lack of domain specific scripting languages for designing background processes that consume Linked Data. To address this weakness, we design a scripting language with a simple but appropriate type system. In our proposed architecture some data is consumed from sources outside of the control of the system and some data is held locally. Stronger type assumptions can be made about the local data than external data, hence our type system mixes static and dynamic typing. Throughout, we relate our work to the W3C recommendations that drive Linked Data, so our syntax is accessible to Web developers.

1 Introduction

Linked data is raw data published on the Web that makes use of URIs to establish links between datasets. The use of URIs to identify resources allows data about resources to be looked up (dereferenced) using a simple protocol, and for the data returned to contain more URIs that can also be looked up. Linked Data consumers can crawl the Web of Linked Data to pull in data that can enrich a Web application. For example the 2012 Olympics Website used Linked Data to help journalists discover and organise statistics about relatively unknown medal winners during the games. Due to links bringing down barriers between datasets, the Web of Linked Data is amongst the worlds largest datasets in the hands of Web developers.

We describe a simple, but appropriate architecture for a Linked Data consumer. We then address the problem of designing a type system for this architecture. Linked Data published on the Web from multiple sources is inherently messy, so data arriving over HTTP must be dynamically type checked. Dynamically type checked data is then loaded into a local triple store. The local triple store persists a view of the Web of Linked Data relevant to the Linked Data consumer. Once the consumed Linked Data has been dynamically typed checked, queries and scripts over the local data can be type checked using a mix of dynamic and static type checking. Finally, because we can never have a global view of the Web of Linked Data, we take care to design our type system so that type checking is local. To achieve this we select a useful subset of the W3C standards SPARQL, RDF Schema and OWL to design our type system.

In Section 2, we describe the Linked Data architecture, with an emphasis on consuming Linked Data. In Section 3 we argue for a notion of type that aligns with the relevant W3C recommendations and is as simple as possible whilst picking up basic programming errors. In Section 4, we define the syntax of our scripting language for consuming Linked Data and the rules of our type system.

A. Ravara and J. Silva (Eds.): 9th Workshop on
Automated Specification and Verification of Web Systems (WWV'13)
EPTCS 123, 2013, pp. 19–33, doi:10.4204/EPTCS.123.4

2 From a Web of Documents to a Web of Data

In 1989, Tim Berners-Lee proposed a hypertext system that became the World Wide Web. In his proposal [2], he observes that hypertext systems from the 1980's failed to gain traction because they attempted to justify themselves on their own data. He described a simple but effective architecture for exposing existing data from file systems and databases as HTML documents that link to other documents using URIs. This architecture is still used today to present documents that link to documents.

Despite the success of the World Wide Web, Berners-Lee was not completely satisfied. He also wanted to make raw data itself Web-like, not just the documents that present data. His first vision was called the Semantic Web [4], which was an AI textbook vision of a world where intelligent agents would understand data on the Web to do every day tasks on our behalf. As admitted by Berners-Lee and his co-author Hendler, there was much hype but limited success scaling ideas. Hendler self-critically asked: "Where are all the intelligent agents?" [19]. By 2006 [30], Berners-Lee had come to the conclusion that there had been too much emphasis on deep ontologies and not enough data.

Thus Berners-Lee returned to the grass roots of the Web: the Web developers. He described a simple protocol for publishing raw data on the Web [3]. The protocol makes use of standards, namely URIs as global identifiers, HTTP as a transport layer and RDF as a data format, according to the following principles:

- use URI to identify resources (i.e. anything that might be referred to in data),
- use HTTP URIs to identify resources so we can look them up (using the HTTP GET verb),
- when a URI is looked up, return data about the resource using the standards (RDF),
- include URIs in the data, so they can also be looked up.

Data published according to the above protocol is called *Linked Data*. An HTTP URI that returns data when it is looked up is a *dereferenceable* URI. All URIs that appear in this paper are dereferenceable, so are part of this rapidly growing Web of Linked Data [18].

The Linked Data protocol is one example of a *RESTful* protocol [14]. A RESTful protocol runs directly on the HTTP protocol using the HTTP verbs (including GET, PUT and DELETE) which are suited to services of publishing data. Many data protocols such as the Twitter API[1], Facebook Open Graph protocol[2] and the Google Data API[3] are RESTful, and with some creativity they can be broadly interpreted as Linked Data. Hence, like the Web of hypertext, the Web of Linked Data does not need to justify itself solely on its own data.

2.1 An Architecture for Consuming Linked Data

Data owners may want to publish their data as Linked Data. Publishing Linked Data is no more difficult than building a traditional Web page. The developer should provide an RDF view of a dataset rather than an HTML view [6]. Data from diverse sources such as Wikipedia [7] and governments [31] can be lifted to the Web of Linked Data.

Data consumers may not own data, but have a data centric service to deliver. Data consumers can consume data from many Linked Data sources then exploit links between datasets. Consuming Linked Data is the main focus in our work. In Figure 1, we describe a simple architecture for an application that consumes Linked Data. The architecture is an extension of the traditional Web architecture.

[1] Twitter API: `https://dev.twitter.com/docs/api/1.1`
[2] Facebook Open Graph protocol: `https://developers.facebook.com/docs/opengraph/`
[3] Google Data API: `https://developers.google.com/gdata/`

Figure 1: A simple but effective architecture for an application that consumes Linked Data.

At the heart of our application is a *triple store*, which replaces the more traditional relational database. A triple store is optimized for storing RDF data in subject-property-object form corresponding to a labelled edge in a graph. There are several commercial grade triple stores including Sesame, Virtuoso and 4store [9, 13, 16], which can operate on scales of billions of triples, i.e. enough for almost any Web application.

The front end of our application follows the traditional Web architecture pattern, where Web pages are generated from a database using scripts. The only difference is that our application uses SPARQL Query instead of SQL to read from the triple store. The syntax of SPARQL is similar to SQL, hence it is easy for an experienced Web developer to develop the front end. Most popular Web development frameworks, such as Ruby on Rails, have been extended to support SPARQL. The main reason that SQL is replaced with SPARQL is that a triple store typically stores data from heterogeneous data sources. Heterogeneous data sources are difficult to combine and query using tables with relational schema; whereas combining and querying graph models is more straightforward. Thus links between datasets can be more easily exploited in queries.

The key novel feature of an application that consumes Linked Data is the back end. At the back end, background processes can crawl the Web of Linked Data to discover new and relevant Linked Data sources. Back end processes also keep local Linked Data up-to-date. For example, data from a news feed may change hourly and changes are made to Wikipedia several times every second. To consume data relevant to the application, the background processes should be programmable. This work focuses in particular on high level programming languages that can be used when programming the back end of an application. The background processes must be able to dereference Linked Data and update data in the triple store, as well as make decisions based on query results.

2.2 A Low Level Approach to Consuming Linked Data

The back end of the application that consumes Linked Data should keep track of every URI accessed through the HTTP protocol. The HTTP header of an HTTP response contains information that can be used for discovering and maintaining Linked Data about a given URI. We describe the dereferencing of URIs at a low level, to be concrete about what a high level language should abstract.

Consider an illustrative example of dereferencing a URI. If we perform an HTTP GET on the URI *dbpedia:Kazakhstan* with an HTTP header indicating that we accept the mime type `text/n3`, then we get a *303 See Other response*.[4] A *303 See Other response* means that you can get data of the serialisation type you requested at another location. If we now perform an HTTP GET request at the URI indicated by the *303 See Other response* (`http://dbpedia.org/data/Kazakhstan.n3`), we get an HTTP 200 OK response including the following headers:

```
GET /data/Kazakhstan.n3 HTTP/1.1      HTTP/1.1 200 OK
Host: dbpedia.org                     Date: Tue, 26 Mar 2013 15:39:49 GMT
Accept: text/n3                       Content-Type: text/n3
```

[4] Reproducible using: `curl -v -I -H "Accept:text/n3" http://dbpedia.org/resource/Kazakhstan`

From the response header (above right) we can tell that this URI successfully returned some RDF using the `n3` serialisation format. However, although the data was obtained from the second URI, the resource represented by *dbpedia:Kazakhstan* is described in the data.

The *303 See Other response* is one of several ways Linked Data may be published using a RESTful protocol [18]. Furthermore, some systems such as the Virtuoso triple store [13] use wrappers to extract RDF data from other data sources, such as the Google Data API or the Twitter API. The programmer of a script that consumes Linked Data should not worry about details such as wrappers or the serialisation formats. Unfortunately, existing libraries for popular programming languages [1, 27] are at a low level of abstraction. We propose that a higher level language can hide the above details in a compiler that tries automatically to dereference URIs.

2.3 A High Level Approach to Consuming Linked Data

Here we consider languages that consume Linked Data at a higher level of abstraction. The only essential information is the URI to dereference and the data returned. Other features of a high level language include control flow and queries to decide what other URIs to dereference.

Consider the following script (based loosely on SPARQL [17]). The keyword `select` binds a term bound to variable `$x`. The `from named` keyword indicates that we dereference the URI indicated. The `where` keyword indicates that we would like to match a given triple pattern. Notice that the second `from named` keyword dereferences a URI bound to `$x` that is not known until the query patter is matched.

```
from named dbpedia:Kazakhstan
select $x
where
   graph dbpedia:Kazakhstan {dbpedia:Kazakhstan  dbp:capital  $x}
from named $x
```

The above script first ensures that data representing *dbpedia:Kazakhstan* is stored in the named graph, also named *dbpedia:Kazakhstan* , in the local triple store. If the URI has not been accessed before, then the URI is dereferenced. If the URI is dereferenced successfully, then the RDF data is stored in a named graph in the local triple store. Thus the data can be queried directly from the named graph in the local triple store. This gives our applications a local view of the Web of Linked Data. Note that, if the URI is not dereferenced successfully, then this information can be recorded to avoid future attempts to dereference the URI.

Having dereferenced the first URI, the query indicated in the `where` clause is evaluated. The query consists of a named graph to read, indicated by the `graph` keyword, and a triple pattern. In the pattern, the subject is the resource *dbpedia:Kazakhstan* , the property is *dbp:capital* , and the object is not bound. This query should find exactly one binding for `$x` to proceed. The query proceeds by discovering a URI (in this case *dbpedia:Astana*), and substituting this URI for `$x` everywhere in the script. After the substitution, the second `from named` keyword dereferences the discovered URI and loads the data into the triple store, as described above.

The above script abstracts away several HTTP GET requests and possibly some redirects and mappings between formats. It also encapsulates details where the following SPARQL Query is sent to the local SPARQL endpoint.

```
select $x from named dbpedia:Kazakhstan where
    { graph dbpedia:Kazakhstan {dbpedia:Kazakhstan  dbp:capital  $x} } limit 1
```

At a lower level of abstraction, the above query would return a results document indicating that `$x` has one possible binding. Another programming language would extract the binding from the results document, then use the binding in some code that dereferences the URI. Just doing this simple task in Java for example takes several pages of code. The Java program also involves treating parts of the code, such as the above query as a raw string of characters, which means that even basic errors parsing the syntax cannot be checked at compile time.

We argue that using the high level script presented at the beginning of this section is simpler than using a general purpose language with libraries concerned with details of HTTP GET requests, constructing queries from strings and extracting variable bindings from query results. Furthermore, it is worth noting that the syntax of the script does not significantly depart from the syntax of the SPARQL recommendations. In this way, we explore the idea of a domain specific scripting language for background processes of an application that consumes Linked Data.

3 Simple Types in W3C Recommendations

The W3C recommendations do not explicitly introduce a type system for Linked Data. However, there are some ideas in the RDF Schema [8] and OWL [20] recommendations that can be used as a basis of a type system. Here we identify one of the simplest notion of a type, and justify the choice with respect to recommendations. The chosen notion of a type fits with types in Facebook's Open Graph.

Simple Datatypes. The only common component of the W3C recommendations in this section is the notion of a simple datatype. Simple datatypes are specified as part of the XML Schema recommendation [5]. A type system for Linked Data should be aware of the simple datatypes that most commonly appear, in particular *xsd:string*, *xsd:integer*, *xsd:decimal* and *xsd:dateTime*. All these types draw from disjoint lexical spaces, except *xsd:integer* is a subtype of *xsd:decimal*. Note that we assume that *xsd:decimal*, *xsd:float* and *xsd:double* are different lexical representations of an abstract numeric type.

In the XML Schema recommendation, there is a simple datatype *xsd:anyURI*. This type is rarely actively used in ontologies – the ontology for DBpedia only uses this type once as the range of the property *dbp:review*. However, it unambiguously refers to any URI and nothing else, unlike *rdfs:Resource* and *owl:Thing* which, depending on the interpretation, may refer to more than just URIs or a subset of URIs. In this way, our type system is based on unambiguous simple datatypes, that frequently appear in datasets, such as DBpedia [7].

Resource Description Framework. A URI is successfully dereferenced when we get a document from which we can extract RDF data [24]. The basic unit of RDF is the *triple*. An RDF triple consists of a subject, a property and an object. The subject, property and object may be URIs, and the object may also be a simple datatype. Most triple stores support *quadruples*, where a fourth URI represents either the *named graph* [10] or the *context* from where the triple was obtained.

Note that the RDF recommendation allows nodes with a local identifier, called a blank node, in place of a URI. Blank nodes are frequently debated in the community [26], due to several problems they introduce. Firstly, deciding the equality of graphs with blank nodes is an NP-complete problem; and, more seriously, when data with blank nodes is consumed more than once, each time the blank node is treated as a new blank node. This can cause many unnecessary copies of the same data to be created. We assume that our system assigns a new URI to each blank node in data consumed, hence do not introduce blank nodes into the local data model.

It is also important to note that the RDF specification has a vocabulary of URIs that have a distinguished role. Notably, the property *rdf:type* is used to indicate a URI that classifies the resource. For example, the triple *dbpedia:Kazakhstan rdf:type dbp:Country* classifies the resource *dbpedia:Kazakhstan* as a *dbp:Country*. Although the word "type" is part of the URI for the property, we consider this triple to be part of the data format rather than part of our type system. In RDF, the word "type" is used in the AI sense of a semantic network [32], rather than in a type theoretic sense. Since these "types" can be changed like any other data, we make the design decision not to include them in our type system, because a type system is used for static analysis.

RDF Schema. The RDF Schema recommendation [8] provides a core vocabulary for classifying resources using RDF. From this vocabulary we borrow only the top level class *rdfs:Resource* and the property *rdfs:range*. All URIs are considered to identify resources, hence we equate *rdfs:Resource* and *xsd:anyURI*. We define property types for URIs that are used in the property position of a triple. A property type restricts the type of term that can be used in the object position of a triple. For example, according to the DBpedia ontology, the property *dbp:populationDensity* has a range *xsd:decimal*. Thus our type system should accept that *dbpedia:Kazakhstan dbp:populationDensity* 5.94 is well typed. However, the type system should reject a triple with object `"5.94"` which is a string. We use the notation *range*(*xsd:decimal*) for URIs representing properties permitting numbers as objects.

The RDF Schema *rdfs:domain* of a property is redundant for our type system because only URIs can appear as the subject of a triple, and all URIs are of type *xsd:anyURI*. Note that properties are resources because they may appear in data. For example, the triple *dbp:populationDensity rdfs:label* `"population density (/sqkm)"@en` provides a description of the property in English.

Web Ontology Language. The Web Ontology Language (OWL) [20] is mostly concerned with classifying resources, which is not part of our type system. The OWL classes that are related to our type system are *owl:ObjectProperty*, *owl:DataTypeProperty* and *owl:Thing*. An *owl:ObjectProperty* is a property permitting URIs as objects, i.e. the type *range*(*xsd:anyURI*) in our type system. An *owl:DataTypeProperty* is a property with one of the simple datatypes as its value.

In OWL [20], *owl:Thing* represents resources that are neither properties nor classes. We decide to equate *owl:Thing* with *xsd:anyURI* in our type system. This way we unify *xsd:anyURI*, *rdfs:Resource* and *owl:Thing* as the top level of all resources. We do not consider any further features of OWL to be part of our type system.

SPARQL Protocol and RDF Query Language. The SPARQL suite of recommendations makes reference only to simple types. SPARQL Query [17] specifies the types of basic operations that are used to filter queries. For example, a regular expression can only apply to a string, and the sum to two integers is an integer. SPARQL Query treats all URIs as being of type URI. We also adopt this approach.

Open Graph Protocol. Facebook's Open Graph protocol uses an approach to Linked Data called microformats, where bits of RDF data are embedded into HTML documents. Microformats help machines to understand the content of the Web pages, which can be used to drive powerful searches. The Open Graph documentation states the following: "properties have 'types' which determine the format of their values."[5] In the terminology of the Open Graph documentation, the value of a property is the object of

[5] `https://developers.facebook.com/docs/opengraph/property-types/` accessed on 27 March 2013.

an RDF triple. The documentation explicitly includes the simple data types *xsd:string*, *xsd:integer*, etc, as types permitted to appear in the object position. This corresponds to the notion of type throughout this section.

3.1 Local Type Checking for Dereferenced Linked Data

We design our system such that, when a URI is dereferenced, only well typed queries are loaded into the triple store. This means that we can guarantee that all triples in the triple store are well typed.

Suppose that after dereferencing the URI *dbpedia:Kazakhstan* we obtain the following two triples:

dbpedia:Kazakhstan dbp:demonym `"Kazakhstani"@en` .
dbpedia:Kazakhstan dbp:demonym dbpedia:Kazakhstani .

Suppose also that the property *dbp:demonym* has the type *range*(*xsd:string*). The first triple is well typed, hence it is loaded into the store in the named graph *dbpedia:Kazakhstan*. However, the second triple is not well typed, hence would be ignored. No knowledge of other triples loaded into the store is required, i.e. our type system is local.

Since only well typed triples are loaded into the local triple store, scripts that use data in the local triple store can rely on type properties. Thus, for example, if a script consumes the object of any triple with *dbp:demonym* as the property, then the script can assume that the term returned will be a string. This allows some static analysis to be performed by a type system for scripts. This observation is the basis of the type system in the next section.

4 A Typed Scripting Language for Linked Data Consumers

In this section, we define our language and type system. A grammar specifies the abstract syntax, and deductive rules specify the type system. We briefly outline the operational semantics, which is defined as a reduction system.

4.1 Syntax

We introduce a syntax for a typed high level scripting language that is used to consume Linked Data.

The Syntax of Types. A type is either a simple datatype, or it is a property that allows a simple datatype as its range, as follows:

$$\mathit{datatype} ::= \mathit{xsd{:}anyURI} \mid \mathit{xsd{:}string} \mid \mathit{xsd{:}decimal} \mid \mathit{xsd{:}dateTime} \mid \mathit{xsd{:}integer}$$

$$\mathit{type} ::= \mathit{datatype} \mid \mathit{range}(\mathit{datatype}) \qquad \mathit{variable} ::= \mathtt{\$x} \mid \mathtt{\$y} \mid \ldots \qquad \Gamma ::= \epsilon \mid \mathtt{\$x}\colon \mathit{type}, \Gamma$$

If a 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. Property types are assigned to URIs using the finite partial function $O(\cdot)$ from URIs to property types. This partial function can be derived from ontologies or inferred from data.

Type environments are defined by lists of assignments of variables to types. As in popular scripting languages, such as Perl and PHP, variables begin with a dollar sign.

The Syntax of Terms and Expressions. Terms are used to construct RDF triples. Terms can be URIs, variables or literals of a simple datatype, each of which is drawn from a disjoint pool of lexemes.

$$\textit{uri} ::= \textit{dbpedia:URI} \mid \ldots \qquad \textit{integer} ::= 99 \mid \ldots \qquad \textit{decimal} ::= 99.9 \mid 0.999e2 \ldots$$

$$\textit{string} ::= \texttt{"WWV2013"} \mid \texttt{"workshop"@en-gb} \ldots \qquad \textit{dateTime} ::= \texttt{2013-06-6T13:00:00+01:00} \mid \ldots$$

$$\textit{language-range} ::= \texttt{*} \mid \texttt{en} \mid \texttt{en-gb} \mid \ldots \qquad \textit{term} ::= \textit{variable} \mid \textit{uri} \mid \textit{string} \mid \textit{integer} \mid \textit{decimal} \mid \textit{dateTime}$$

$$\textit{regex} ::= \texttt{WWV.*} \mid \ldots \qquad \textit{expr} ::= \textit{term} \mid \texttt{now} \mid \texttt{str}(\textit{expr}) \mid \texttt{abs}(\textit{expr}) \mid \textit{expr} + \textit{expr} \mid \textit{expr} - \textit{expr} \mid \ldots$$

Notice that strings may have a language tag, as defined by RFC4646 [28]. A language tag can be matched by a simple pattern, called a language range, where * matches any language tag (e.g., en matches any dialect of English). Regular expressions over strings conform to the XPath recommendation [25].

Expressions are formed by applying unary and binary functions over terms. The SPARQL Query recommendation [17] defines several standard functions including str, which maps any term to a string, and abs, which takes the absolute value of a number. The expression now represents the current date and time. The vocabulary of functions may be extended as required.

The Syntax of Scripts. Scripts are formed from boolean expressions, data and queries. They define a sequence of operations that use queries to determine what URIs to dereference.

$$\begin{aligned} \textit{boolean} ::= {} & \textit{boolean} \;||\; \textit{boolean} \mid \textit{boolean} \;\&\&\; \textit{boolean} \mid \neg \textit{boolean} \\ \mid {} & \texttt{regex}(\textit{expr}, \textit{regex}) \mid \texttt{langMatches}(\textit{expr}, \textit{language-range}) \mid \textit{expr} = \textit{expr} \mid \textit{expr} < \textit{expr} \mid \ldots \end{aligned}$$

$$\textit{triples} ::= \textit{term term term} \mid \textit{triples triples} \qquad \textit{data} ::= \texttt{graph}\,\textit{term}\,\{\textit{triples}\} \mid \textit{data data}$$

$$\textit{query} ::= \textit{data} \mid \textit{boolean} \mid \textit{query query} \mid \textit{query}\,\texttt{union}\,\textit{query}$$

script ::=	`where` *query script*	*Satisfy a query pattern before continuing.*
|	`from named` *term script*	*Dereference a URI and load it into the local triple store.*
|	`select` *variable* : *type script*	*Select a binding for a variable to enable progress.*
|	`do` *script*	*Iteratively execute the script using separate data.*
|	`skip`	*Successfully terminate.*

Data is represented as quadruples of terms, which always indicate the named graph. In SPARQL, the graph and from named keywords work in tandem. The from named keyword makes data from a named graph available, whilst keeping track of the context. The keyword graph allows the query to directly refer to the context. This contrasts to the from keyword in SPARQL which fetches data without keeping the context. We extend the meaning of the from named keyword so that the URI is dereferenced and makes the data available from that point onwards in the context of the URI that is referenced.

Boolean expressions are called filters in the terminology of SPARQL. We drop the keyword filter, because the syntax is unambiguous without it. Filters can be used to match a string with a regular expression or language tag, or compare two expressions of the same type.

Queries are constructed from filters and basic graph patterns indicated by the keyword where. The basic graph pattern should be matched using data in the local triple store. The basic graph pattern may contain variables. Variables in a script must be bound using the select keyword. In this scripting

language, the `select` keyword acts like an existential quantifier. The variables bound by `select` are annotated with a type. However these type annotations may be omitted by the programmer, since they can be algorithmically inferred using the type system.

4.2 The Type System

For existing implementations of SPARQL, a query that violates types silently fails, returning an empty result. However, a type error is generally caused by an oversight by the programmer. It would be helpful if a type error is provided at compiler time, indicating that a query has been designed such that the types guarantee that no result will ever be returned. For this purpose, we introduce the type system presented in this section.

Subtypes. We define a subtype relation, that defines when one type can be treated as another type. The system indicates that *xsd:integer* is a subtype of *xsd:decimal*. It also defines property types to be contravariant, i.e. they reverse the direction of subtyping. In particular, if a property permits decimal numbers in the object position, then it also permits integers in the object position.

$$\frac{}{\vdash \mathit{xsd{:}integer} \leq \mathit{xsd{:}decimal}} \qquad \frac{}{\vdash \mathit{type} \leq \mathit{type}}$$

$$\frac{\vdash \mathit{datatype}_1 \leq \mathit{datatype}_2}{\vdash \mathit{range}(\mathit{datatype}_2) \leq \mathit{range}(\mathit{datatype}_1)} \qquad \frac{}{\vdash \mathit{range}(\mathit{datatype}) \leq \mathit{xsd{:}anyURI}}$$

The subtype relations between types that can be assigned to URIs are summarised in Figure 2.

The type system for terms and expressions. Types for terms assign types to lexical tokens and assign types to properties using the partial function $O(\cdot)$ from URIs to types. Types for expressions ensure that

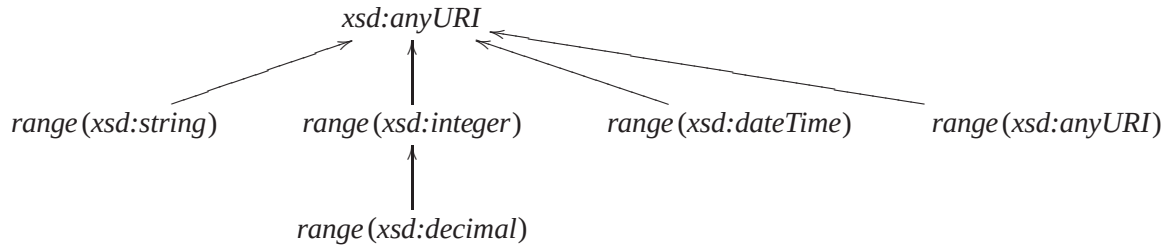

Figure 2: Subtype relations between types that can be assigned to URIs.

operations are only applied to resources of the correct type.

$$\frac{\vdash type_0 \leq type_1}{\Gamma, \$x\colon type_0 \vdash \$x\colon type_1} \qquad \frac{\vdash O(uri) \leq type}{\Gamma \vdash uri\colon type} \qquad \frac{\vdash xsd{:}integer \leq datatype}{\Gamma \vdash integer\colon datatype}$$

$$\frac{}{\Gamma \vdash decimal\colon xsd{:}decimal} \qquad \frac{}{\Gamma \vdash string\colon xsd{:}string} \qquad \frac{}{\Gamma \vdash dateTime\colon xsd{:}dateTime}$$

$$\frac{}{\Gamma \vdash \texttt{now}\colon xsd{:}dateTime} \qquad \frac{\Gamma \vdash expr_1\colon datatype \quad \Gamma \vdash expr_2\colon datatype \quad \vdash datatype \leq xsd{:}decimal}{\Gamma \vdash expr_1 + expr_2\colon datatype}$$

$$\frac{\Gamma \vdash expr\colon datatype}{\Gamma \vdash \texttt{str}(expr)\colon xsd{:}string} \qquad \frac{\Gamma \vdash expr\colon datatype \quad \vdash datatype \leq xsd{:}decimal}{\Gamma \vdash \texttt{abs}(expr)\colon datatype}$$

The above types can easily be extended to cover all functions in the SPARQL recommendation, such as `seconds` which maps an *xsd:dateTime* to a *xsd:decimal*. Our examples include the custom function `haversine` which maps four expressions of type *xsd:decimal* to one *xsd:decimal*.

The type system for filters follows a similar pattern to expressions. For example, the type system ensures that only terms of the same type can be compared.

$$\frac{\Gamma \vdash expr\colon xsd{:}string}{\Gamma \vdash \texttt{regex}(expr, regex)} \qquad \frac{\Gamma \vdash expr\colon xsd{:}string}{\Gamma \vdash \texttt{langMatches}(expr, language\text{-}range)}$$

$$\frac{\Gamma \vdash expr_1\colon datatype \quad \Gamma \vdash expr_2\colon datatype}{\Gamma \vdash expr_1 = expr_2} \qquad \frac{\Gamma \vdash expr_1\colon datatype \quad \Gamma \vdash expr_2\colon datatype}{\Gamma \vdash expr_1 < expr_2}$$

$$\frac{\Gamma \vdash boolean_0 \quad \Gamma \vdash boolean_1}{\Gamma \vdash boolean_0 \;\&\&\; boolean_1} \qquad \frac{\Gamma \vdash boolean_0 \quad \Gamma \vdash boolean_1}{\Gamma \vdash boolean_0 \;||\; boolean_1} \qquad \frac{\Gamma \vdash boolean}{\Gamma \vdash !boolean}$$

The type system for data. The types for terms are used to restrict the subject of triples to URIs, and the object to the type prescribed by the property. For quadruples, the named graph is always a URI, as prescribed by the type system.

$$\frac{\Gamma \vdash term_1\colon xsd{:}anyURI \quad \Gamma \vdash term_2\colon range(datatype) \quad \Gamma \vdash term_3\colon datatype}{\Gamma \vdash term_1 \;\; term_2 \;\; term_3}$$

$$\frac{\Gamma \vdash triples_1 \quad \Gamma \vdash triples_2}{\Gamma \vdash triples_1 \; triples_2} \qquad \frac{\Gamma \vdash term_0\colon xsd{:}anyURI \quad \Gamma \vdash triples}{\Gamma \vdash \texttt{graph}\, term_0\, \{triples\}}$$

$$\frac{\Gamma \vdash term\colon range(datatype)}{\Gamma \vdash term\ rdfs{:}range\ datatype\ .} \qquad \frac{\Gamma \vdash term\colon range(xsd{:}anyURI)}{\Gamma \vdash term\ rdf{:}type\ owl{:}ObjectProperty\ .}$$

We include special type rules for two particular forms of triples. The first, from the RDF Schema vocabulary [8], allows the range of a property to be explicitly prescribed as a datatype. The second, from the OWL vocabulary [20], prescribes when the range of a property is a URI. These two rules are useful for future work in type inference. In particular, when type information is not available, these rules would help infer the minimum partial function $O(\cdot)$ that allows a dataset to be typed.

The Type System for scripts. Scripts can be type checked using our type system. The rule for the `from named` keyword checks that the term to dereference is a URI. The rule for the `select` makes use of an assumption in the type environment to ensure that the variable is used consistently in the rest of the script, where the variable is bound.

$$\frac{\Gamma \vdash \mathit{query}_1 \quad \Gamma \vdash \mathit{query}_2}{\Gamma \vdash \mathit{query}_1\ \mathit{query}_2} \qquad \frac{\Gamma \vdash \mathit{query}_1 \quad \Gamma \vdash \mathit{query}_2}{\Gamma \vdash \mathit{query}_1\ \texttt{union}\ \mathit{query}_2} \qquad \frac{\Gamma \vdash \mathit{query} \quad \Gamma \vdash \mathit{script}}{\Gamma \vdash \texttt{where}\ \mathit{query}\ \mathit{script}}$$

$$\frac{\Gamma \vdash \mathit{term} : \mathit{xsd{:}anyURI} \quad \Gamma \vdash \mathit{script}}{\Gamma \vdash \texttt{from named}\ \mathit{term}\ \mathit{script}} \qquad \frac{\Gamma, \texttt{\$x} : \mathit{type} \vdash \mathit{script}}{\Gamma \vdash \texttt{select \$x}\ : \mathit{type}\ \mathit{script}} \qquad \frac{\Gamma \vdash \mathit{script}}{\Gamma \vdash \texttt{do}\ \mathit{script}} \qquad \Gamma \vdash \texttt{skip}$$

If a script is well typed with an empty type environment, then all the variables must be bound using the rule for the `select` quantifier. Furthermore, since the `select` quantifiers does not appear in data, no variables can appear in well typed data. We assume that scripts and data are executable only if they are well typed with respect to an empty type environment.

4.3 Examples of Well Typed Scripts

We consider some well typed scripts, and suggest some errors that our type system avoids.

The following script is well typed. The script finds resources in any named graph that have a label in the Russian language. It then dereferences the resources. The script is iterated as many times as the implementation feels necessary, without revisiting data.

```
do  select $g: xsd:anyURI, $x: xsd:anyURI, $y: xsd:string
    where
       graph $g {$x rdfs:label $y}
       langMatches($y, ru)
    from named $x
```

Note that if we use the variable `$y` instead of `$x` in the `from named` clause, then the script could not be typed. The variable `$y` would need to be both a string and a URI, but it can only be one or the other. We assume that $\mathcal{O}(\mathit{rdfs{:}label}) = \mathit{range}(\mathit{xsd{:}string})$.

The following well typed script looks in two named graphs. In named graph *dbpedia:Kazakhstan*, it looks for properties with *dbpedia:Kazakhstan* as the object, and in the named graph *dbp:* it looks for properties that have either a label or comment that contains the string `"location"`.

```
select $p: range(xsd:anyURI), $y: xsd:string, $z: xsd:string
where
   {
     graph dbp: {$p rdfs:label $y}
     union
     graph dbp: {$p rdfs:comment $y}
   }
   graph dbpedia:Kazakhstan {$z $p dbpedia:Kazakhstan}
   regex($y, location) && langMatches($y, en)
from named $p
```

We must assume that $O(\textit{rdfs:comment}) = \textit{range}(\textit{xsd:string})$. If we assume otherwise, then the above query is not well typed. Note also that property `$p` must be of type $\textit{range}(\textit{xsd:anyURI})$. We cannot assign it a more general type such as *xsd:anyURI* , although we can use it as a resource.

```
from named dbpedia:Almaty
select $almalat: xsd:decimal, $almalong: xsd:decimal
where
   graph dbpedia:Almaty {dbpedia:Almaty geo:lat $almalat}
   graph dbpedia:Almaty {dbpedia:Almaty geo:long $almalong}
from named dbpedia:Kazakhstan
do select $loc: xsd:anyURI
   where
      graph dbpedia:Kazakhstan {$loc dbp:location dbpedia:Kazakhstan}
   from named $loc
   select $lat: xsd:decimal, $long: xsd:decimal
   where
      graph $loc {$loc geo:lat $lat}
      graph $loc {$loc geo:long $long}
      haversine($lat, $long, $almalat, $almalong) < 100
   do select $person: xsd:anyURI
      where
         graph $loc {$person dbp:birthPlace $loc}
      from named $person
```

Figure 3: Get data about people born in places in Kazakhstan less than 100km from Almaty.

Finally, consider the substantial example of Figure 3. We assume that the function `haversine` calculates the distance (in km) between two points identified by their latitude and longitude. The script pulls in data about places located in Kazakhstan. It then uses this data to pull in more data about people born in places less than 100km from Almaty.

4.4 Operational Semantics for the System

In related work, we extensively study the operational semantics of languages for Linked Data that are related to the language proposed in this work [11, 12, 21, 22, 23]. In the remaining space, we briefly sketch the operational semantics for our scripting language.

Systems are data and scripts composed in parallel by using the $\|$ operator. The main rules are the rules for dereferencing URIs, for selecting bindings, and for interactions between a query and data. The rules can be applied in any context.

$$\frac{\vdash \texttt{graph}\, \textit{uri}\, \{\textit{triples}\}}{\texttt{from named}\, \textit{uri}\, \textit{script} \longrightarrow \textit{script} \,\|\, \texttt{graph}\, \textit{uri}\, \{\textit{triples}\}}$$

$$\frac{\vdash \textit{term} : \textit{type}}{\texttt{select \$x}: \textit{type}\, \textit{script} \longrightarrow \textit{script}\left\{{}^{\textit{term}}/_{\$x}\right\}} \qquad \frac{\textit{data} \leq \textit{query}}{\texttt{where}\, \textit{query}\, \textit{script} \,\|\, \textit{data} \longrightarrow \textit{script} \,\|\, \textit{data}}$$

The rules for dereferencing data involves a dynamic type check of arbitrary data that arrives as a result of dereferencing a URI. The `select` rule also performs a dynamic check to ensure that the term substituted for a variable is of the correct type. The query rule reads data that matches the query pattern (using a preorder $\leq$ over queries), without removing the data.

The following result proves that a well typed term will always reduce to a well typed term. Thus the type system is sound with respect to the operational semantics.

Theorem 4.1. *If* $\vdash$ *system*$_1$ *and system*$_1 \longrightarrow$ *system*$_2$, *then* $\vdash$ *system*$_2$.

Proof. We provide a proof sketch covering the cases for only the three rules above.

Consider the operational rule for `from named`. Assume that both $\vdash$ `from named` *uri script* and $\vdash$ `graph` *uri* {*triples*} hold. By the type rule for `from named`, $\vdash$ *script* must hold. Thus, by the type rule for parallel composition, $\vdash$ *script* $\|$ `graph` *uri* {*triples*}.

Lemma. If $\vdash$ *term*: *type* and `$x`: *type* $\vdash$ *script*, then $\vdash$ *script*$\{^{term}/_{\$x}\}$, by structural induction. □

Consider the operational rule for `select`. Assume that $\vdash$ *term*: *type* and $\vdash$ `select $x`: *type script* hold. By the type rule for `select`, `$x`: *type* $\vdash$ *script* must hold. Hence, by the above lemma, $\vdash$ *script*$\{^{term}/_{\$x}\}$ holds.

Consider the operational rule for `where`. Assume that $\vdash$ `where` *query script* $\|$ *data* holds. By the type rule for parallel composition, $\vdash$ `where` *query script* and $\vdash$ *data* must hold, and, by the type rule for `where`, $\vdash$ *query* and $\vdash$ *script* must hold. Hence $\vdash$ *script* $\|$ *data* holds. □

Further cases and results will be covered in an extended paper. Future work includes implementing an interpreter for the language based on the operational semantics, and developing a minimal type inference algorithm [29] based on the type system.

5 Conclusion

As the Web of Linked Data grows, the state of the art for commercial Linked Data solutions is also advancing. State of the art of triple stores allow efficient execution of queries at scale. Furthermore, the back end of commercial solutions such as Virtuoso [13] and the Information Workbench [15] can extract data from diverse sources. This allows us to take a liberal view of the Web of Data, where data is drawn from data APIs provided by popular services from Twitter, Facebook and Google. Through experience with master students, we found that developers with experience of a Web development platform such as .NET or Ruby-on-Rails can assemble a front end in a matter of days, simply by shifting their query language from SQL to SPARQL.

Linked Data enables processes to programmatically crawl the Web of Linked Data, pulling data from diverse sources and removing boundaries between datasets. The data pulled from the Web forms a local view of the Web of Linked Data that is tailored to a particular application. We found that existing programming environments, consisting of a general purpose language and a library, obstructed swift development of such processes with many low level details. This exposes the need for a high level language that makes scripting background processes that consume Linked Data easy.

In this work, we introduce a domain specific high level language for consuming Linked Data. Domain specific languages are designed at a level of abstraction that simplifies programming tasks in the domain. In our domain specific language, key operations such as queries are primitive, meaning that basic syntactic checks can be performed. It is also easier to perform static analysis over a domain specific

language. We take care to design the syntax of the scripting language such that it resembles the SPARQL recommendations [17], to appeal to the target Web developers.

We introduce a simple but effective type system for our language. The type system is based on the fragment of the SPARQL, RDF Schema and OWL recommendations that deals with simple data types. The applications can statically identify simple errors such as attempts to dereference a number, or attempts to match the language tag of a URI. For static type checking of scripts, the data loaded into the system must be dynamically type checked to ensure that properties have the correct literal value or a URI as the object. The dynamic type checks do not impose significant restrictions on the data consumed. Most datasets, including data from DBpedia, conform to this typing pattern. Further weight is added by the Facebook Open Graph protocol, which demands typing at exactly the level we deliver.

Acknowledgements. The work was supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, project number PN-II-ID-PCE-2011-3-0919.

References

[1] David Beckett (2002): *The design and implementation of the Redland RDF application framework*. *Computer Networks* 39(5), pp. 577–588, doi:10.1016/S1389-1286(02)00221-9.

[2] Tim Berners-Lee (1989): *Information management: A proposal*. Available at http://www.w3.org/History/1989/proposal.html.

[3] Tim Berners-Lee (2006): *Linked data — design issues*. Available at http://www.w3.org/DesignIssues/LinkedData.html.

[4] Tim Berners-Lee, James Hendler & Ora Lassila (2001): *The Semantic Web*. *Scientific American* 284(5), pp. 28–37, doi:10.1038/scientificamerican0501-34.

[5] Paul V. Biron & Ashok Malhotra (2004): *XML Schema part 2: Datatypes Second Edition*. Recommendation REC-xmlschema-2-20041028, MIT, Cambridge, MA. Available at http://www.w3.org/TR/xmlschema-2/.

[6] Christian Bizer & Andy Seaborne (2004): *D2RQ-treating non-RDF databases as virtual RDF graphs*. In: *Proceedings of the 3rd International Semantic Web Conference (ISWC2004)*, p. 26, doi:10.1038/npre.2011.5660.1.

[7] Christian Bizer et al. (2009): *DBpedia: A Crystallization Point for the Web of Data*. *Web Semantics: Science, Services and Agents on the World Wide Web* 7(3), pp. 154–165, doi:10.1016/j.websem.2009.07.002.

[8] Dan Brickley & Ramanathan V. Guha (2004): *RDF Vocabulary Description Language 1.0: RDF Schema*. recommendation REC-rdf-schema-20040210, W3C, MIT, Cambridge, MA. Available at http://www.w3.org/TR/rdf-schema/.

[9] Jeen Broekstra, Arjohn Kampman & Frank van Harmelen (2002): *Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema*. In: *International Semantic Web Conference*, pp. 54–68, doi:10.1007/3-540-48005-6_7.

[10] Jeremy J. Carroll, Christian Bizer, Pat Hayes & Patrick Stickler (2005): *Named graphs*. *Web Semantics: Science, Services and Agents on the World Wide Web* 3(4), pp. 247–267, doi:10.1016/j.websem.2005.09.001.

[11] Gabriel Ciobanu & Ross Horne (2012): *A Provenance Tracking Model for Data Updates*. In: *FOCLASA*, pp. 31–44, doi:10.4204/EPTCS.91.3.

[12] Mariangiola Dezani, Ross Horne & Vladimiro Sassone (2012): *Tracing where and who provenance in Linked Data: a calculus*. *Theoretical Computer Science* 464, pp. 113–129, doi:10.1016/j.tcs.2012.06.020.

[13] Orri Erling & Ivan Mikhailov (2007): *RDF Support in the Virtuoso DBMS*. In: *CSSW*, pp. 59–68, doi:10.1007/978-3-642-02184-8_2.

[14] Roy T. Fielding & Richard N. Taylor (2002): *Principled design of the modern Web architecture*. *ACM Transactions on Internet Technology* 2(2), pp. 115–150, doi:10.1145/514183.514185.

[15] Peter Haase, Michael Schmidt Christian Hütter & Andreas Schwarte (2012): *The Information Workbench as a Self-Service Platform for Linked Data Applications*. In: *WWW 2012. Lyon, France*.

[16] Steve Harris, Nick Lamb & Nigel Shadbolt (2009): *4store: The design and implementation of a clustered RDF store*. In: *5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, pp. 94–109.

[17] Steve Harris & Andy Seaborne (2013): *SPARQL 1.1 Query Language*. Recommendation REC-sparql11-query-20130321, W3C, MIT, MA. Available at http://www.w3.org/TR/sparql11-query/.

[18] Tom Heath & Christian Bizer (2011): *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web, Morgan & Claypool Publishers. Available at http://dx.doi.org/10.2200/S00334ED1V01Y201102WBE001.

[19] James A. Hendler (2007): *Where Are All the Intelligent Agents? IEEE Intelligent Systems* 22(3), pp. 2–3. Available at http://doi.ieeecomputersociety.org/10.1109/MIS.2007.62.

[20] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider & Sebastian Rudolph (2012): *OWL 2 Web Ontology Language Primer (Second Edition)*. Recommentation REC-owl2-primer-20121211, W3C, MIT, Cambridge, MA. Available at www.w3.org/TR/owl2-primer/.

[21] Ross Horne & Vladimiro Sassone (2011): *A Verified Algebra for Linked Data*. In: *FOCLASA*, pp. 20–33, doi:10.4204/EPTCS.58.2.

[22] Ross Horne & Vladimiro Sassone (2013): *A Verified Algebra for Read-Write Linked Data*. *Science of Computer Programming*. To appear.

[23] Ross Horne, Vladimiro Sassone & Nicholas Gibbins (2012): *Operational Semantics for SPARQL Update*. In: *The Semantic Web, Lecture Notes in Computer Science* 7185, Springer, pp. 242–257, doi:10.1007/978-3-642-29923-0_16.

[24] Graham Klyne & Jeremy Carroll (2004): *Resource Description Framework: Concepts and Abstract Syntax*. recommendation REC-rdf-concepts-20040210, W3C, MIT, MA. Available at http://www.w3.org/TR/rdf-concepts/.

[25] Ashok Malhotra, Jim Melton, Norman Walsh & Michael Kay (2010): *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)*. recommendation REC-xpath-functions-20101214, W3C, MIT, MA. Available at www.w3.org/TR/xpath-functions/.

[26] Alejandro Mallea, Marcelo Arenas, Aidan Hogan & Axel Polleres (2011): *On Blank Nodes*. In: *International Semantic Web Conference (1)*, pp. 421–437, doi:10.1007/978-3-642-25073-6_27.

[27] Brian McBride (2002): *Jena: A Semantic Web Toolkit*. *IEEE Internet Computing* 6(6), pp. 55–59, doi:10.1109/MIC.2002.1067737.

[28] A. Phillips & M. Davis (2006): *Tags for Identifying Languages*. Best Current Practice RFC4646, IETF. Available at http://www.ietf.org/rfc/rfc4646.

[29] Michael I Schwartzbach (1991): *Type inference with inequalities*. In: *Proceedings of the international joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (CAAP '91): vol 1*, TAPSOFT '91, Springer, pp. 441–455.

[30] Nigel Shadbolt, Wendy Hall & Tim Berners-Lee (2006): *The Semantic Web revisited*. *IEEE intelligent systems* 21(3), pp. 96–101, doi:10.1109/MIS.2006.62.

[31] Nigel Shadbolt, Kieron O'Hara, Tim Berners-Lee, Nicholas Gibbins, Hugh Glaser, Wendy Hall & m. c. schraefel (2012): *Linked Open Government Data: Lessons from Data.gov.uk*. *IEEE Intelligent Systems* 27(3), pp. 16–24, doi:10.1109/MIS.2012.23.

[32] John F. Sowa (2000): *Knowledge representation: logical, philosophical and computational foundations*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA.