Minimal Type Inference for Linked Data Consumers [1]

Gabriel Ciobanu[a], Ross Horne[b], Vladimiro Sassone[c]

[a] *Romanian Academy, Institute of Computer Science, Blvd. Carol I, no. 8, 700505 Iaşi, Romania*
[b] *Faculty of Information Technology, Kazakh British Technical University, Almaty, Kazakhstan*
[c] *Electronics and Computer Science, University of Southampton, United Kingdom*

Abstract

We provide an introduction to the Web of Linked Data from the perspective of a Web developer who would like to build an application using Linked Data. We identify a weakness in the development stack, namely 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 about external data, hence our type system mixes static and dynamic typing. We prove that our type system is algorithmic; and can therefore be used for minimal type inference. We also prove subject reduction and type safety results, which justify our claim that our language is statically type checked and does not throw basic runtime type errors. Throughout, we relate our work to the W3C recommendations that drive Linked Data, so that our syntax is accessible to Web developers.

Keywords: type systems, operational semantics, Linked Data

1. Introduction

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. In data published as Linked Data [6], we use URIs (Web addresses) to identify resources to allow data about resources to be dereferenced (looked up) using a simple protocol. The data obtained by dereferencing URIs contains more URIs that can also be looked up. The advantage of this model is that, like the Web of hypertext, there are no boundaries between datasets. A dataset can unambiguously refer to a URI used as an identifier in any other dataset.

In the world of Linked Data you can be a producer or a consumer. A Linked Data producer typically owns data, e.g. public government data [48], that would benefit from being openly published so that it can be combined with other published datasets. A Linked Data consumer typically has a service to deliver that can be enriched by data from external sources. For example, the Web site for the 2012 Olympics used Linked Data to help journalists discover and organise statistics about little-known medal winners during the games. Due to links bringing down barriers between datasets, the Web of Linked Data is establishing itself as one of the world's largest datasets in the hands of Web developers. In this work, we focus on the requirements of Web developers that consume Linked Data.

The state of the art for commercial Linked Data solutions enable rich Linked Data driven Web sites to be developed. The core of an application that consumes Linked Data is a triple store. Commercial grade triple stores [13, 22, 28] allow efficient execution of queries over loosely structured data at scales of billions of triples, which is sufficient to cache enough data for most applications. Column store based techniques enable warehouse scale analytical workloads over triples [1]. Furthermore, the back end of commercial solutions such as Virtuoso [22] and the Information Workbench [27] can extract data from diverse sources. This allows us to take a liberal view of the Web of Data, where

[1]©2015. This manuscript version is made available under the CC-BY-NC-ND 4.0 license. Article available on ScienceDirect at DOI:10.1016/j.jlamp.2014.12.005. Corresponding author: Ross Horne.
Email address: `ross.horne@gmail.com` (Ross Horne)

data can be extracted from open 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 for a Linked Data driven application in a matter of days, simply by shifting their query language from SQL to SPARQL.

In contrast to our positive experience with triple stores and front end development, we found that existing programming environments that consist of a general purpose language and a library obstructed swift development of background processes with low level details. Background process are processes that are not called directly by clients, but instead run independently in the background automatically discovering and updating data in the triple store. This exposes the need for a high level language that simplifies the process of scripting background processes that consume Linked Data. The first contribution of this work is to specify exactly such a scripting language.

We then address the problem of designing a type system for our scripting language. Linked Data published on the Web from multiple sources is inherently messy, so data arriving over HTTP must be dynamically type checked. The 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 type checked, queries and scripts over the local data can be type checked using a mix of dynamic and static type checking. Our type system is designed to respect a small but useful subset of the W3C standards SPARQL, RDF Schema and OWL.

The objective of this work is not to define an elaborate or novel type system. On the contrary, the objective is to design a language and type system that are as simple as possible, while still being useful to Linked Data consumers. The runtime errors avoided by the type system, such as dereferencing a number or adding a string to a number, routinely occur when programming. Our type system avoids these errors with minimal impact on the programmer. Indeed, due to our minimal type inference algorithm, the programmer need not be aware of the type system.

In the area of Linked Data, considerable attention has been paid to "types" in the AI sense of semantic networks [49]. However, there is little work on types in Linked Data in the type theoretic sense. A more elaborate type system that covers a larger part of the RDF Schema recommendation [11] appears in [33], but at the cost of blurring the distinction between types and data. Elsewhere, another notion of type is employed in [20], for enforcing access control policies on Linked Data based on provenance. This paper is an extended version of a workshop paper [18] presented at WWV 2013. This paper extends the workshop paper with results that show that the type system is algorithmic, hence can be used for type inference. This paper also features more comprehensive subject reduction and type safety results. The final contribution is a discussion on how the type system can be used to infer schema information, which in Linked Data is not provided a priori.

Structure. In Section 2, we describe a simple architecture for a Linked Data consumer that allows sufficiently strong assumptions to be made about the types of locally persisted 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. We prove that our type system is algorithmic, and hence can be used for type inference. In Section 5, we define a operational semantics for our language based on reductions. The operational semantics are used to prove the correctness of our type system, via subject reduction and type safety results. In Section 6, we discuss how our type system can assist in extracting and inferring schema information. Finally, we discuss related work on types, languages and consumming data from the Web in Section 7.

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 [5], 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 [7], 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?" [31]. By 2006 [47], 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 [6]. 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 URIs 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 [30].

The Linked Data protocol is one example of a *RESTful* protocol [23], which leverages the HTTP verbs (including GET, PUT and DELETE) to consume and publish resources identified by URIs. Many data protocols such as the Twitter API[2], Facebook Open Graph protocol[3] and the Google Data API[4] are RESTful, and with some creativity they can be broadly interpreted as Linked Data. Hence, like the Web of hypertext, the Web of Data should not 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 [10]. Data from diverse sources such as Wikipedia [9] and governments [48] 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.

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 [13, 22, 28], which can operate at scales of billions of triples, i.e. enough for typical Web applications.

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

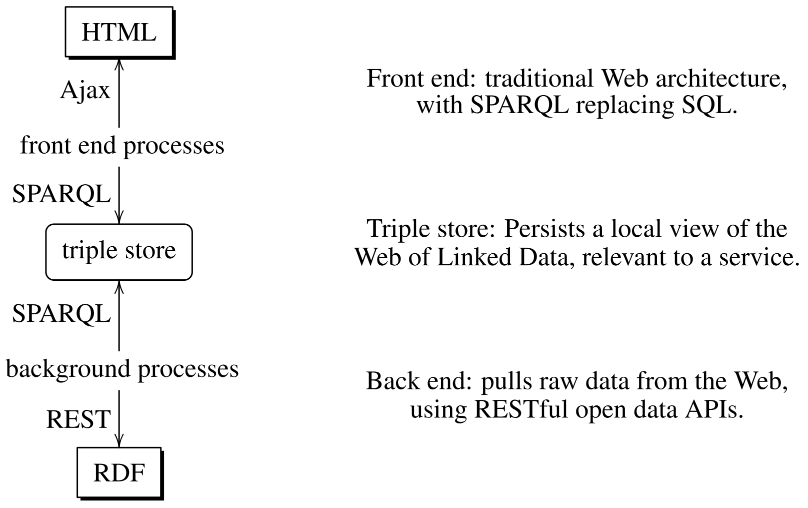
HTML
Ajax
front end processes
SPARQL
triple store
SPARQL
background processes
REST
RDF
Front end: traditional Web architecture, with SPARQL replacing SQL.
Triple store: Persists a local view of the Web of Linked Data, relevant to a service.
Back end: pulls raw data from the Web, using RESTful open data APIs.

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

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. 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. To dereference URI *dbpedia:Kazakhstan* we perform an HTTP GET with an HTTP header indicating that we accept the mime type `text/n3`, in which case we get a *303 See Other* response similar to the following.[5]

```
GET /resource/Kazakhstan HTTP/1.1      HTTP/1.1 303 See Other
Host:  dbpedia.org                     Content-Type:  text/n3
Accept:  text/n3                       Location:  http://dbpedia.org/data/Kazakhstan.n3
```

A *303 See Other response* means that you can get data of the serialisation type you requested at another location, indicated by the location header in the response above. 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
```

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 [30]. Furthermore, some systems such as the Virtuoso triple store [22] 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 [4, 39] are at a low level of abstraction. We propose that a higher level language [50, 24] can hide the above details in a compiler that automatically tries to dereference URIs.

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

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 [29]). The `from named` keyword indicates that we dereference the URI indicated. The keyword `select` binds a term to the variable `$x`. The `where` keyword indicates that we would like to match a given triple pattern. Notice that the second `from named` keyword dereferences a URI bound to `$x` that is not known until the query pattern 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 the resource *dbpedia:Kazakhstan* is stored in a distinguished sub graph with a name, called a *named graph* [14]. It is good practice to create a new named graph in the local triple store for every distinct URI dereferenced. Each named graph is named according to the URI that was dereferenced — in this case the URI *dbpedia:Kazakhstan.* This gives our applications a partial view of the Web of Linked Data in their local triple store, which can be queried with low latency.

If the URI has not been accessed before, then the URI is dereferenced, and a new named graph is created. 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 syntax errors 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 [11] and OWL [32] 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 [8]. 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, with the exception of *xsd:integer* which is a subtype of *xsd:decimal*. Note that, for this work, 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 explicitly 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 [9].

Resource Description Framework. A URI is successfully dereferenced when we get a document from which we can extract RDF data [19]. 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* [14] 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 [38], 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 we do not introduce blank nodes into the local data model. URIs can be assigned to blank nodes in a number of ways. For example, the consumer may generate a fresh URI for each blank node in a suitable namespace, where the URI generated encodes unique provenance information about the blank node.

Note that the RDF specification has a vocabulary of URIs that have a distinguished role, notably *rdf:type*. For example, the triple *dbpedia:Kazakhstan rdf:type dbpedia:Republic* classifies the resource *dbpedia:Kazakhstan* as a *dbpedia:Republic*. The property *rdf:type* has little to do with the simple data types in this work. In RDF, *rdf:type* is used in the AI sense of a semantic network [49], rather than in a type theoretic sense. Triples that use the property *rdf:type* can be changed like any other data, e.g. a country can change from a *dbpedia:Republic* to a *dbpedia:Federation* . However, a type system is used for static analysis. Hence we make the design decision not to include AI types in our type system. For comparison, [34] allows more information from AI style types to be lifted to the type system.

RDF Schema. The RDF Schema recommendation [11] 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) [32] is mostly concerned with classifying resources, but classifying resources 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 [32], *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 [29] 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 that determine the format of their values."[6] In the terminology of the Open Graph documentation, the value of a property is the object of 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 triple store and put in the named graph *dbpedia:Kazakhstan*. However, the second triple is not well typed, and would therefore 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:

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

$$\mathit{type} ::= \mathit{datatype} \mid \mathit{range}(\mathit{datatype}) \qquad \Gamma ::= \epsilon \mid \texttt{\$x}\colon \mathit{type}, \Gamma$$

[6] `https://developers.facebook.com/docs/opengraph/property-types/` accessed 23/3/13.

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. URIs used as properties are assigned a datatype using the finite partial function $\mathcal{S}(\cdot)$ from URIs to datatypes, called the *schema*. The data type determines the type of term that can occur in the object position. The schema can be derived from ontologies, such as the DBpedia ontology, or inferred from how the property is used in data. Note that constructing and maintaining $\mathcal{S}(\cdot)$ can be challenging — a point we return to in Section 6.

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. Thus the token *variable* ranges over the lexemes `$x`, `$y`, etc.

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[7].

- URIs, indicated by token *uri*, are either full addresses, defined by RFC3987 [21], surrounded by angular brackets, e.g. ⟨*http://dbpedia.org/resource/URI*⟩, or abbreviated using an XML namespace, e.g. *dbpedia:URI*.
- Numbers are either integers or decimals, indicated by tokens *integer* and *decimal*. Decimals can be either floating point numbers, e.g. 99.9, or floating point numbers with an exponent, e.g. $0.999e2$.
- Strings, indicated by token *string*, e.g. `"verification"@en-GB`, are a list of characters in quotes followed by a language tag indicating the language used in the string. Language tags are defined by RFC4646 [43].
- The token *dateTime* ranges over lexemes that encode the date, time in hours, minutes and seconds and time zone, e.g. `2013-06-06T13:00:00+01:00`.

A language tag can be matched by a simple pattern, called a language range (*lang-range*), where `*` matches any language tag, `en` matches any dialect of English, etc. Language ranges are specified in RFC4646 [43].

A string can be accepted or rejected by a regular expression. For example the regular expression `WWV.*` accepts the string `"WWV 2013"`. Regular expressions over strings conform to the XPath recommendation [37].

Expressions, defined by the grammar below, are formed by applying unary and binary functions over terms.

$$\mathit{term} ::= \mathit{variable} \mid \mathit{uri} \mid \mathit{string} \mid \mathit{integer} \mid \mathit{decimal} \mid \mathit{dateTime}$$

$$\mathit{expr} ::= \mathit{term} \mid \texttt{now} \mid \texttt{str}(\mathit{expr}) \mid \texttt{abs}(\mathit{expr}) \mid \mathit{expr} + \mathit{expr} \mid \mathit{expr} - \mathit{expr} \mid \ldots$$

The SPARQL Query recommendation [29] defines several standard functions including `str`, which maps any term to a string, and `abs`, which returns the absolute value of a number. The expression `now` represents the current date and time. The vocabulary of functions may be extended as required.

[7] Lexical concerns are hidden in high level languages, e.g. the short syntax for turtle [45]. At a low level, a lexer [2] would convert the lexeme 3 into a attribute-value/token-name pair `"3"^^xsd:integer`, as in [19].

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.

```
boolean ::= true                                    triples ::= term term term
          | false                                             | triples triples
          | boolean || boolean
          | boolean && boolean
          | !boolean
          | regex (expr, regex)                     data ::= graph term {triples}
          | langMatches(expr, lang-range)                  | data data
          | expr = expr                                    | {}
          | expr < expr | ...

                                         script ::= where query script
query ::= data                                    | from named term script
        | boolean                                 | select variable: type script
        | query query                             | do script
        | query union query                       | script script
                                                  | success
```

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

According to the SPARQL recommendation, a query that violates types silently fails, returning an empty result. However, a type error is generally caused by an oversight of the programmer. It would be helpful to provide a type error at compile 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.

$$\begin{tikzcd}
& \mathit{xsd{:}anyURI} & & \\
\mathit{range}(\mathit{xsd{:}string}) \arrow[ur] & \mathit{range}(\mathit{xsd{:}integer}) \arrow[u] & \mathit{range}(\mathit{xsd{:}dateTime}) \arrow[ul] & \mathit{range}(\mathit{xsd{:}anyURI}) \arrow[ull] \\
& \mathit{range}(\mathit{xsd{:}decimal}) \arrow[u] & &
\end{tikzcd}$$

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

Subtypes. We define a subtype relation, that decides 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. Types for URIs that are used as properties are determined using the *schema* partial function $\mathcal{S}(\cdot)$, which determines the range of the property, i.e. what terms can appear in the object position of a triple. Types for expressions ensure that operations are only applied to resources of the correct type.

$$\frac{\vdash \mathit{type}_0 \leq \mathit{type}_1}{\Gamma, \texttt{\$x}\colon \mathit{type}_0 \vdash \texttt{\$x}\colon \mathit{type}_1} \qquad \frac{\vdash \mathit{range}(\mathcal{S}(\mathit{uri})) \leq \mathit{type}}{\Gamma \vdash \mathit{uri}\colon \mathit{type}} \qquad \frac{\vdash \mathit{xsd{:}integer} \leq \mathit{datatype}}{\Gamma \vdash \mathit{integer}\colon \mathit{datatype}}$$

$$\frac{}{\Gamma \vdash \mathit{uri}\colon \mathit{xsd{:}anyURI}} \qquad \frac{}{\Gamma \vdash \mathit{decimal}\colon \mathit{xsd{:}decimal}} \qquad \frac{}{\Gamma \vdash \mathit{string}\colon \mathit{xsd{:}string}}$$

$$\frac{}{\Gamma \vdash \mathit{dateTime}\colon \mathit{xsd{:}dateTime}} \qquad \frac{}{\Gamma \vdash \texttt{now}\colon \mathit{xsd{:}dateTime}}$$

$$\frac{\Gamma \vdash \mathit{expr}_1\colon \mathit{datatype} \quad \Gamma \vdash \mathit{expr}_2\colon \mathit{datatype} \quad \vdash \mathit{datatype} \leq \mathit{xsd{:}decimal}}{\Gamma \vdash \mathit{expr}_1 + \mathit{expr}_2\colon \mathit{datatype}}$$

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

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

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

$$\frac{\Gamma \vdash \mathit{expr}_1\colon \mathit{datatype} \quad \Gamma \vdash \mathit{expr}_2\colon \mathit{datatype}}{\Gamma \vdash \mathit{expr}_1 = \mathit{expr}_2} \qquad \frac{\Gamma \vdash \mathit{expr}_1\colon \mathit{datatype} \quad \Gamma \vdash \mathit{expr}_2\colon \mathit{datatype}}{\Gamma \vdash \mathit{expr}_1 < \mathit{expr}_2}$$

$$\frac{\Gamma \vdash \mathit{boolean}_0 \quad \Gamma \vdash \mathit{boolean}_1}{\Gamma \vdash \mathit{boolean}_0 \mathbin{\&\&} \mathit{boolean}_1} \qquad \frac{\Gamma \vdash \mathit{boolean}_0 \quad \Gamma \vdash \mathit{boolean}_1}{\Gamma \vdash \mathit{boolean}_0 \mathbin{||} \mathit{boolean}_1} \qquad \frac{\Gamma \vdash \mathit{boolean}}{\Gamma \vdash !\mathit{boolean}}$$

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.

$$\frac{\Gamma \vdash \mathit{term}_1 : \mathit{xsd{:}anyURI} \quad \Gamma \vdash \mathit{term}_3 : \mathcal{S}(\mathit{uri})}{\Gamma \vdash \mathit{term}_1 \ \ \mathit{uri} \ \ \mathit{term}_3}$$

$$\frac{\Gamma, \texttt{\$x} : \mathit{range}(\mathit{datatype}) \vdash \mathit{term}_1 : \mathit{xsd{:}anyURI} \quad \Gamma, \texttt{\$x} : \mathit{range}(\mathit{datatype}) \vdash \mathit{term}_3 : \mathit{datatype}}{\Gamma, \texttt{\$x} : \mathit{range}(\mathit{datatype}) \vdash \mathit{term}_1 \ \ \texttt{\$x} \ \ \mathit{term}_3}$$

$$\frac{\Gamma \vdash \mathit{triples}_1 \quad \Gamma \vdash \mathit{triples}_2}{\Gamma \vdash \mathit{triples}_1 \ \mathit{triples}_2} \qquad \frac{\Gamma \vdash \mathit{term}_0 : \mathit{xsd{:}anyURI} \quad \Gamma \vdash \mathit{triples}}{\Gamma \vdash \texttt{graph}\, \mathit{term}_0 \,\{\, \mathit{triples} \,\}}$$

$$\frac{\vdash \mathit{datatype} \leq \mathcal{S}(\mathit{uri})}{\vdash \mathit{uri} \ \ \mathit{rdfs{:}range} \ \ \mathit{datatype}} \qquad \frac{\vdash \mathit{xsd{:}anyURI} \leq \mathcal{S}(\mathit{uri}_0)}{\vdash \mathit{uri}_0 \ \ \mathit{rdfs{:}range} \ \ \mathit{uri}_1} \qquad \frac{\vdash \mathit{xsd{:}anyURI} \leq \mathcal{S}(\mathit{uri})}{\vdash \mathit{uri} \ \ \mathit{rdf{:}type} \ \ \mathit{owl{:}ObjectProperty}}$$

We include special type rules for three particular forms of triples. The first, from the RDF Schema vocabulary [11], allows the range of a property to be explicitly prescribed as a datatype. The second implies that if the range of a property is anything other than a datatype, then the property must range over URIs. The second, from the OWL vocabulary [32], prescribes when the range of a property is a URI. When schema information is not available about URIs used as properties, these rules can help calculate the minimum schema partial function $\mathcal{S}(\cdot)$ that allows a dataset to be typed. We return to these rules in Section 6, where we explain how schema information can be extracted using the type system.

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.

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

$$\Gamma \vdash \texttt{success} \qquad \frac{\Gamma \vdash \mathit{query} \quad \Gamma \vdash \mathit{script}}{\Gamma \vdash \texttt{where}\ \mathit{query}\ \mathit{script}} \qquad \frac{\Gamma \vdash \mathit{term} : \mathit{xsd{:}anyURI} \quad \Gamma \vdash \mathit{script}}{\Gamma \vdash \texttt{from named}\ \mathit{term}\ \mathit{script}}$$

$$\frac{\Gamma, \texttt{\$x} : \mathit{type} \vdash \mathit{script}}{\Gamma \vdash \texttt{select \$x} : \mathit{type}\ \mathit{script}} \qquad \frac{\Gamma \vdash \mathit{script}}{\Gamma \vdash \texttt{do}\ \mathit{script}} \qquad \frac{\Gamma \vdash \mathit{script}_0 \quad \Gamma \vdash \mathit{script}_1}{\Gamma \vdash \mathit{script}_0\ \mathit{script}_1}$$

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` quantifier 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 has a label in the Russian language. It then dereferences the resources. The keyword do means that the script is iterated, where the number of times a script is iterated should correspond to the number of query results.

```
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 $\mathcal{S}$ (*rdfs:label*) = *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 $\mathcal{S}$ (*rdfs:comment*) = *xsd:string*. If we assume that the range of *rdfs:comment* is anything other than *xsd:string*, then the above query is not well typed. Note also that property `$p` must be of type *range*(*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 within 100km from Almaty.

4.4. Algorithmic Typing

The type system for our scripting language can be used as the basis of a minimal type inference algorithm [44]. Type inference reverses the reading of the type system. For type checking, we are given an environment Γ and script *script* with type annotations, and we ask whether we can prove the type judgement $\Gamma \vdash$ *script*. In contrast, for minimal type inference, given a script without type annotations *script*, we ask whether we can find the weakest type environment Γ and most general type annotation script' such that $\Gamma \vdash \mathit{script}'$.

To use a type system in a type inference algorithm, we must check that the type system is algorithmic: every type rule should use only subterms of its conclusion (the formula below the line) in its premises (the formulae above the line). If this property holds, then the type system is syntax directed.

A subsumption rule and the transitivity rule for subtyping would violate the subformula property. It is customary to prove the following lemmas, to show that if the transitivity or subsumption rule were included, then they can be eliminated from the type system.

Lemma 4.1. *If* $\vdash \mathit{type}_0 \leq \mathit{type}_1$ *and* $\vdash \mathit{type}_1 \leq \mathit{type}_2$*, then* $\vdash \mathit{type}_0 \leq \mathit{type}_2$.

Proof. The trick is to introduce a transitivity rule into the subtype system, then show that it can be eliminated. The base case, presented below, is when the reflexivity axiom appears on the left or right branch of the transitivity rule. In this case, the reflexivity axiom destroys the transitivity rule, as follows.

$$\dfrac{\dfrac{}{\vdash \mathit{type}_0 \leq \mathit{type}_0} \qquad \vdash \mathit{type}_0 \leq \mathit{type}_1}{\vdash \mathit{type}_0 \leq \mathit{type}_1} \qquad \text{yields} \qquad \vdash \mathit{type}_0 \leq \mathit{type}_1$$

Consider the case where the anyURI axiom appears as the right branch of a transitivity rule, as follows.

$$\dfrac{\vdash \mathit{range}(\mathit{datatype}_0) \leq \mathit{range}(\mathit{datatype}_1) \qquad \dfrac{}{\vdash \mathit{range}(\mathit{datatype}_1) \leq \mathit{xsd{:}anyURI}}}{\vdash \mathit{range}(\mathit{datatype}_0) \leq \mathit{xsd{:}anyURI}}$$

The left branch above can be removed, leaving only the axiom below.

$$\dfrac{}{\vdash \mathit{range}(\mathit{datatype}_0) \leq \mathit{xsd{:}anyURI}}$$

Consider the case where transitivity is applied to two property types, as follows.

$$\dfrac{\dfrac{\vdash \mathit{datatype}_1 \leq \mathit{datatype}_2}{\vdash \mathit{range}(\mathit{datatype}_2) \leq \mathit{range}(\mathit{datatype}_1)} \qquad \dfrac{\vdash \mathit{datatype}_0 \leq \mathit{datatype}_1}{\vdash \mathit{range}(\mathit{datatype}_1) \leq \mathit{range}(\mathit{datatype}_0)}}{\vdash \mathit{range}(\mathit{datatype}_2) \leq \mathit{range}(\mathit{datatype}_0)}$$

The transitivity rule can be pushed up the derivation tree and applied to the datatypes, resulting in the derivation tree below.

$$\dfrac{\dfrac{\vdash \mathit{datatype}_0 \leq \mathit{datatype}_1 \qquad \vdash \mathit{datatype}_1 \leq \mathit{datatype}_2}{\vdash \mathit{datatype}_0 \leq \mathit{datatype}_2}}{\vdash \mathit{range}(\mathit{datatype}_2) \leq \mathit{range}(\mathit{datatype}_0)}$$

By induction, since $\vdash \mathit{datatype}_0 \leq \mathit{datatype}_1$ and $\vdash \mathit{datatype}_1 \leq \mathit{datatype}_2$ hold, we can assume that $\vdash \mathit{datatype}_0 \leq \mathit{datatype}_2$ can be derived without using transitivity. This covers all possible cases. Hence the transitivity rule can be eliminated. □

Lemma 4.2. *If* $\Gamma \vdash \mathit{expr} \colon \mathit{type}_0$ *and* $\vdash \mathit{type}_0 \leq \mathit{type}_1$ *hold, then* $\Gamma \vdash \mathit{expr} \colon \mathit{type}_1$ *holds.*

Proof. The trick is to introduce the subsumption rule, below, then prove that it can be eliminated.

$$\dfrac{\vdash \Gamma \vdash \mathit{expr} \colon \mathit{type}_0 \qquad \vdash \mathit{type}_0 \leq \mathit{type}_1}{\vdash \Gamma \vdash \mathit{expr} \colon \mathit{type}_1}$$

The base cases are the axioms for terms. Consider the case of variables. Assuming that $\vdash \mathit{type}_0 \leq \mathit{type}_1$ and $\vdash \mathit{type}_1 \leq \mathit{type}_2$ hold, by Lemma 4.1, $\vdash \mathit{type}_0 \leq \mathit{type}_2$ holds. Hence given the derivation below on the left, we can construct the derivation below on the right.

$$\dfrac{\dfrac{\vdash \mathit{type}_0 \leq \mathit{type}_1}{\vdash \Gamma, \mathtt{\$x}\colon \mathit{type}_0 \vdash \mathtt{\$x}\colon \mathit{type}_1} \quad \vdash \mathit{type}_1 \leq \mathit{type}_2}{\vdash \Gamma, \mathtt{\$x}\colon \mathit{type}_0 \vdash \mathtt{\$x}\colon \mathit{type}_2} \qquad\qquad \dfrac{\vdash \mathit{type}_0 \leq \mathit{type}_2}{\vdash \Gamma, \mathtt{\$x}\colon \mathit{type}_0 \vdash \mathtt{\$x}\colon \mathit{type}_2}$$

The cases for simple data types and URIs are similar.

Consider the rule for the addition of two numbers, as follows.

$$\dfrac{\dfrac{\Gamma \vdash \mathit{expr}_0\colon t_0 \quad \Gamma \vdash \mathit{expr}_1\colon t_0 \quad \vdash t_0 \leq \mathit{xsd{:}decimal}}{\Gamma \vdash \mathit{expr}_0 + \mathit{expr}_1\colon t_0} \qquad \vdash t_0 \leq t_1}{\Gamma \vdash \mathit{expr}_0 + \mathit{expr}_1\colon t_1}$$

The only two possible values for t_1 are *xsd:integer* and *xsd:decimal*, hence $t_1 \leq \mathit{xsd{:}decimal}$. Thus we can construct the following derivation.

$$\dfrac{\dfrac{\Gamma \vdash \mathit{expr}_0\colon t_0 \quad \vdash t_0 \leq t_1}{\Gamma \vdash \mathit{expr}_0\colon t_1} \quad \dfrac{\Gamma \vdash \mathit{expr}_1\colon t_0 \quad \vdash t_0 \leq t_1}{\Gamma \vdash \mathit{expr}_1\colon t_1} \quad \vdash t_1 \leq \mathit{xsd{:}decimal}}{\Gamma \vdash \mathit{expr}_0 + \mathit{expr}_1\colon t_1}$$

Other cases are either immediate or follow a similar pattern. Hence, by structural induction, we can eliminate the subsumption rule. □

The minimal type inference algorithm works as follows for a script without type annotations:

1. Firstly, we annotate variables appearing in `select` statements with type variables ranging over X, Y, etc.

2. We then apply the rules of the algorithmic type system to generate inequality constraints over type variables. We obtain a system of inequality constraints of the form $\mathit{type}_0 \leq \mathit{type}_1$, where type_0 and type_1 may contain type variables.

3. Finally, we solve the system of inequality constraints. Either we find the assignment of type variables to types such that the constraints are satisfied and the types are maximal with respect to the subtype preorder, or there is no assignment of type variables to types that satisfies the constraints. If there is a solution, then the assignment is the *minimal type assignment* for the script. If there is no assignment, then the script cannot be typed and should be rejected.

The system of inequality constraints is solvable in PTIME [51]. Furthermore, the algorithmic type system is syntax directed, hence the generation of constraints is linear. Therefore our type inference algorithm is in PTIME.

Consider the first example script in Section 4.3 and remove the type annotations. We demonstrate the minimal type inference algorithm for this example in Figure 4.

5. Subject Reduction and Type Safety

To prove the correctness of a type system for a language, we require an operational semantics. An operational semantics is an executable model of the possible behaviours of the language. In this section, we provide a *subject reduction* theorem, which proves that if a well typed program is executed then the program remains well typed. We also provide a *type safety* theorem which proves that well-typed programs do not throw runtime errors.

In related work, we have studied the operational semantics for languages for Linked Data that are related to the language proposed in this work. The calculus in [36] provides the first operational semantics for the transaction language SPARQL Update [25]. The calculus in [35] studies the algebraic properties of transaction languages for

1. Use the algorithmic type system to generate constraints:

$$
\begin{prooftree}
\AxiomC{$G \leq \mathit{xsd{:}anyURI}$}
\UnaryInfC{$\texttt{\$g}\colon G \vdash g\colon \mathit{xsd{:}anyURI}$}
\AxiomC{$X \leq \mathit{xsd{:}anyURI}$}
\UnaryInfC{$\texttt{\$x}\colon X \vdash \texttt{\$x}\colon \mathit{xsd{:}anyURI}$}
\AxiomC{$Y \leq \mathcal{S}\,(\mathit{rdfs{:}label})$}
\UnaryInfC{$\texttt{\$y}\colon Y \vdash \texttt{\$y}\colon \mathit{xsd{:}string}$}
\BinaryInfC{$\texttt{\$x}\colon X, \texttt{\$y}\colon Y \vdash \texttt{\$x}\ \mathit{rdfs{:}label}\ \texttt{\$y}$}
\BinaryInfC{$\texttt{\$g}\colon G, \texttt{\$x}\colon X, \texttt{\$y}\colon Y \vdash \texttt{graph \$g \{\$x}\ \mathit{rdfs{:}label}\ \texttt{\$y\}}$}
\AxiomC{$Y \leq \mathit{xsd{:}string}$}
\UnaryInfC{$\texttt{\$y}\colon Y \vdash \texttt{\$y}\colon \mathit{xsd{:}string}$}
\UnaryInfC{$\texttt{\$y}\colon Y \vdash \texttt{langMatches(\$y, ru-*)}$}
\BinaryInfC{$\texttt{\$g}\colon G, \texttt{\$x}\colon X, \texttt{\$y}\colon Y \vdash \texttt{graph \$g \{\$x}\ \mathit{rdfs{:}label}\ \texttt{\$y\} langMatches(\$y, ru-*)}$}
\AxiomC{$X \leq \mathit{xsd{:}anyURI}$}
\UnaryInfC{$\texttt{\$x}\colon X \vdash \texttt{\$x}\colon \mathit{xsd{:}anyURI}$}
\UnaryInfC{$\texttt{\$x}\colon X \vdash \texttt{from named \$x}$}
\BinaryInfC{$\texttt{\$g}\colon G, \texttt{\$x}\colon X, \texttt{\$y}\colon Y \vdash \texttt{where graph \$g \{\$x}\ \mathit{rdfs{:}label}\ \texttt{\$y\} langMatches(\$y, ru-*) from named \$x}$}
\UnaryInfC{$\vdash \texttt{select \$g}\colon G, \texttt{\$x}\colon X, \texttt{\$y}\colon Y\ \texttt{where graph \$g \{\$x}\ \mathit{rdfs{:}label}\ \texttt{\$y\} langMatches(\$y, ru-*) from named \$x}$}
\UnaryInfC{$\vdash \texttt{do select \$g}\colon G, \texttt{\$x}\colon X, \texttt{\$y}\colon Y\ \texttt{where graph \$g \{\$x}\ \mathit{rdfs{:}label}\ \texttt{\$y\} langMatches(\$y, ru-*) from named \$x}$}
\end{prooftree}
$$

Generated constraints:

$$X \leq \mathit{xsd{:}anyURI} \qquad Y \leq \mathit{xsd{:}string} \qquad G \leq \mathit{xsd{:}anyURI}$$

2. Find the most general solution to the generated constraints [51]:

$$X \mapsto \mathit{xsd{:}anyURI}, \quad Y \mapsto \mathit{xsd{:}string}, \quad G \mapsto \mathit{xsd{:}anyURI}$$

Substituting type variables for the the most general solution results in a well typed annotated script:

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

Figure 4: Example of using the minimal type inference algorithm.

Linked Data. The calculus in [20] introduces security types based on where and who provenance for dereferenced Linked Data. The calculus in [17] models why and how provenance for transactions over Linked Data. Finally, there is existing work that proves subject reduction theorems for type systems inspired by RDF Schema [33]; however this work deliberately selects a smaller and more useful subset of RDF Schema, as explained in Section 3.

5.1. A Syntax for Systems

The behaviour of background processes is influenced by: the scripts themselves, the data in the triple store, and the dereferenced data consumed from the Web. We gather the scripts and data in a triple store together into a system. We introduce some syntax for the purpose of modelling systems.

$$process ::= data \mid script \mid process \parallel process \qquad system ::= [process]_{graphs}$$

Processes are scripts and data composed in parallel. As standard for processes, we assume that parallel composition forms a commutative monoid with the unit `success`.

Systems are processes delimited by brackets and annotated by a set of URIs, called *graphs*. The set of URIs keeps track of the named graphs that have been dereferenced by the system. It is important to delimit the system, since to satisfy three of the following requirements we must have knowledge about all the graphs dereferenced by the system:

- If we evaluate `from named` and the URI indicated has not been dereferenced before, then we dereference the URI. The data can be of any form, so should first be dynamically type checked before being loaded into the triple store.
- If we evaluate `from named` and the URI has already been dereferenced, then we proceed without dereferencing the URI again.
- At any point a URI that has already been dereferenced can be dereferenced again to update the data. This is for maintenance purposes, and the rate should be quantified [16] using stochastic methods so that URIs are revisited at a suitable rate. As with dereferencing the URI for the first time, data coming from the Web should be dynamically type checked.
- If we try to dereference anything other than a URI, then we throw an error.

The four rules below address each of the requirements above. Notice that the first three rules require a system context to determine what graphs have been dereferenced, but the fourth rule does not require knowledge of the rest of the system. When a rule can be applied in any well formed context, we omit the context for brevity.

$$\frac{uri \notin g \qquad \vdash \texttt{graph}\, uri\, \{triples\}}{[\texttt{from named}\; uri\; script \parallel Process]_g \longrightarrow [script \parallel \texttt{graph}\, uri\, \{triples\} \parallel Process]_{g \cup \{uri\}}}$$

$$\frac{uri \in g}{[\texttt{from named}\; uri\; script \parallel Process]_g \longrightarrow [script \parallel Process]_g}$$

$$\frac{uri \in g \qquad \vdash \texttt{graph}\, uri\, \{triples'\}}{[\texttt{graph}\, uri\, \{triples\} \parallel Process]_g \longrightarrow [\texttt{graph}\, uri\, \{triples'\} \parallel Process]_g}$$

$$\frac{not\_a\_uri(term)}{\texttt{from named}\; term\; script \longrightarrow \texttt{error}}$$

Control flow in scripts is based on query results, over data held locally by the system. The `select` quantifier selects a term to replace the variable it binds. A small dynamic check is required to ensure that the term used in the substitution is of the type expected by the select quantifier. The do operator iterates a script zero or more times. At the implementation level, the number of iterations corresponds the number of distinct query results in the query it precedes. The following rules specify the operational behaviour of `select` and `do`.

$$\frac{\vdash \mathit{term}\colon \mathit{type}}{\texttt{select \$x}\colon \mathit{type}\ \mathit{script} \longrightarrow \mathit{script}\left\{{}^{\mathit{term}}/_{\texttt{\$x}}\right\}}$$

$$\frac{}{\texttt{do}\,\mathit{script} \longrightarrow \mathit{script}\ \texttt{do}\,\mathit{script}} \qquad \frac{}{\texttt{do}\,\mathit{script} \longrightarrow \texttt{success}}$$

Note that a poor choice of substitution in the rule for `select` above can result in the script becoming stuck when results are available. Thus this rule can involve some backtracking. Similarly, there can be some backtracking in the rules for evaluating `union` below.

The union keyword in SPARQL, on a result by result basis, is a choice between two query patterns. Either we can chose the left branch or the right branch, as in the first two rules below. When a query is reduced to data only and the data appears in the system, then the conditions of the `where` clause are met and the script can proceed, as indicated by the final rule below.

$$\frac{}{\mathit{query}_0\ \texttt{union}\ \mathit{query}_1 \longrightarrow \mathit{query}_0} \qquad \frac{}{\mathit{query}_0\ \texttt{union}\ \mathit{query}_1 \longrightarrow \mathit{query}_1}$$

$$\frac{}{\texttt{where}\,\mathit{data}\ \mathit{script} \parallel \mathit{data} \longrightarrow \mathit{script} \parallel \mathit{data}} \qquad \frac{}{\texttt{success}\ \mathit{script} \longrightarrow \mathit{script}}$$

Boolean filters can appear in queries to further restrain queries. The logical connectives and, or and not are evaluated according the rules of classical logic. For a modern term rewriting system approach to the semantics of classical propositional logic, consider the system SKS [26].

We assume that we have some standard builtin functions. The functions *accepts* and *matches* decide whether or not a string is accepted by the given regular expression, and whether or not a language-tag matches a language-range. The functions *sum_int* and *sum_dec* add two integers and decimal numbers respectively. The functions *int_to_dec* and *to_str* coerce an integer to a decimal number and any term to a string, respectively. Using the above built-in functions, we can provide an operational semantics for functions and predicates over expressions.

$$\frac{\mathit{accepts}(\mathit{string}, \mathit{regex})}{\texttt{regex}\,(\mathit{string}@\mathit{lang\text{-}tag}, \mathit{regex}) \longrightarrow \texttt{true}} \qquad \frac{\neg\mathit{accepts}(\mathit{string}, \mathit{regex})}{\texttt{regex}\,(\mathit{string}@\mathit{lang\text{-}tag}, \mathit{regex}) \longrightarrow \texttt{false}}$$

$$\frac{\mathit{matches}(\mathit{lang\text{-}tag}, \mathit{lang\text{-}range})}{\texttt{langMatches}(\mathit{string}@\mathit{lang\text{-}tag}, \mathit{lang\text{-}range}) \longrightarrow \texttt{true}}$$

$$\frac{\neg\mathit{matches}(\mathit{lang\text{-}tag}, \mathit{lang\text{-}range})}{\texttt{langMatches}(\mathit{string}@\mathit{lang\text{-}tag}, \mathit{lang\text{-}range}) \longrightarrow \texttt{false}}$$

$$\frac{\mathit{integer}_2 = \mathit{sum\_int}(\mathit{integer}_0, \mathit{integer}_1)}{\mathit{integer}_0 + \mathit{integer}_1 \longrightarrow \mathit{integer}_2} \qquad \frac{\mathit{decimal}_2 = \mathit{sum\_dec}(\mathit{decimal}_0, \mathit{decimal}_1)}{\mathit{decimal}_0 + \mathit{decimal}_1 \longrightarrow \mathit{decimal}_2}$$

$$\frac{\mathit{decimal}_0 = \mathit{int\_to\_dec}(\mathit{integer})}{\mathit{integer} + \mathit{decimal}_1 \longrightarrow \mathit{decimal}_0 + \mathit{decimal}_1} \qquad \frac{\mathit{decimal}_1 = \mathit{int\_to\_dec}(\mathit{integer})}{\mathit{decimal}_0 + \mathit{integer} \longrightarrow \mathit{decimal}_0 + \mathit{decimal}_1}$$

$$\frac{\mathit{integer} < 0}{\texttt{abs}(\mathit{integer}) \longrightarrow -\mathit{integer}} \qquad \frac{0 \leq \mathit{integer}}{\texttt{abs}(\mathit{integer}_0) \longrightarrow \mathit{integer}} \qquad \frac{}{\texttt{str}(\mathit{term}) \longrightarrow \mathit{to\_str}(\mathit{term})}$$

The errors for regular expressions, language matches and operators over numbers are as follows. Functions and predicates over expressions are sensitive to the type of token they are applied to, thus several runtime errors can occur at this stage. For detecting dynamic type errors, we assume the built-in functions *not_a_number*, *not_a_uri* and *not_a_string*, which hold if a term of the wrong token type appears.

$$\frac{\mathit{not\_a\_string}(\mathit{term})}{\texttt{regex}\,(\mathit{term}, \mathit{regex}) \longrightarrow \texttt{error}} \qquad \frac{\mathit{not\_a\_string}(\mathit{term})}{\texttt{langMatches}(\mathit{term}, \mathit{lang\text{-}range}) \longrightarrow \texttt{error}}$$

$$\frac{\mathit{not\_a\_number}(\mathit{term}_0)}{\mathit{term}_0 + \mathit{term}_1 \longrightarrow \texttt{error}} \qquad \frac{\mathit{not\_a\_number}(\mathit{term})}{\texttt{abs}\,(\mathit{term}) \longrightarrow \texttt{error}}$$

5.2. Subject Reduction

We claim that this type system is a novel mix of static and dynamic type checking. Subject reduction is required to justify the claim that the static type checking can be relied upon. The proof works by case analysis on all operational rules.

Lemma 5.1. *If* $\vdash \mathit{term}\colon \mathit{type}$ *and* $\Gamma, \texttt{\$x}\colon \mathit{type} \vdash \mathit{script}$*, then* $\Gamma \vdash \mathit{script}\{^{\mathit{term}}/_{\texttt{\$x}}\}$.

Proof. The proof follows by structural induction on the type derivation tree. The base cases to check are those where a variable is substituted for another term.

Assume that $\vdash \mathit{integer}\colon \mathit{datatype}_0$ and $\Gamma, \texttt{\$x}\colon \mathit{datatype}_0 \vdash \texttt{\$x}\colon \mathit{datatype}_1$ hold. By the type rules for integers and variables, below, it must be the case that $\mathit{xsd{:}integer} \leq \mathit{datatype}_0$ and $\mathit{datatype}_0 \leq \mathit{datatype}_1$ hold.

$$\frac{\vdash \mathit{xsd{:}integer} \leq \mathit{datatype}_0}{\vdash \mathit{integer}\colon \mathit{datatype}_0} \qquad \frac{\vdash \mathit{datatype}_0 \leq \mathit{datatype}_1}{\Gamma, \texttt{\$x}\colon \mathit{datatype}_0 \vdash \texttt{\$x}\colon \mathit{datatype}_1}$$

By Lemma 4.1, $\vdash \mathit{integer} \leq \mathit{datatype}_1$ holds. Hence, by the type rule for integers, the derivation below must hold.

$$\frac{\vdash \mathit{xsd{:}integer} \leq \mathit{datatype}_1}{\Gamma \vdash \mathit{integer}\colon \mathit{datatype}_1}$$

A similar argument works for terms of other terms: strings, decimals, dateTimes and URIs.

The proof then proceeds by structural induction for expressions, booleans, queries and scripts. We present the case for `from named`. Assume that $\vdash \mathit{term}_1\colon \mathit{type}$, that $\Gamma, \texttt{\$x}\colon \mathit{type} \vdash \mathit{term}_0\colon \mathit{xsd{:}anyURI}$ and $\Gamma, \texttt{\$x}\colon \mathit{type} \vdash \mathit{script}$. Hence the following derivation holds.

$$\frac{\Gamma, \texttt{\$x}\colon \mathit{type} \vdash \mathit{term}_0\colon \mathit{xsd{:}anyURI} \quad \Gamma, \texttt{\$x}\colon \mathit{type} \vdash \mathit{script}}{\Gamma, \texttt{\$x}\colon \mathit{type} \vdash \texttt{from named}\ \mathit{term}_0\ \mathit{script}}$$

By structural induction, $\Gamma \vdash \mathit{term}_0\{^{\mathit{term}_1}/_{\texttt{\$x}}\}\colon \mathit{xsd{:}anyURI}$ and $\Gamma \vdash \mathit{script}\{^{\mathit{term}_1}/_{\texttt{\$x}}\}$. Hence we can construct the following type derivation tree.

$$\frac{\Gamma \vdash \mathit{term}_0\left\{^{\mathit{term}_1}/_{\texttt{\$x}}\right\}\colon \mathit{xsd{:}anyURI} \quad \Gamma \vdash \mathit{script}\left\{^{\mathit{term}_1}/_{\texttt{\$x}}\right\}}{\Gamma \vdash (\texttt{from named}\ \mathit{term}_0\ \mathit{script})\left\{^{\mathit{term}_1}/_{\texttt{\$x}}\right\}}$$

Other rules are similar. Thus, by structural induction, the lemma holds. □

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 5.2 (subject reduction)**.** *If* $\vdash \mathit{system}_1$ *and* $\mathit{system}_1 \longrightarrow \mathit{system}_2$*, then* $\vdash \mathit{system}_2$.

Proof. Consider the first operational rule for `from named`. Assume that both the following hold:

$$\vdash [\texttt{from named}\ \mathit{uri}\ \mathit{script} \parallel \mathit{process}]_{\mathit{graph}} \qquad \text{and} \qquad \vdash \texttt{graph}\,\mathit{uri}\,\{\mathit{triples}\}$$

By the type rules for systems and processes and `from named`, it must be the case that $\vdash$ *process* and $\vdash$ *script* hold. Thus, by the type rule for processes and systems, $\vdash [\mathit{script} \parallel \texttt{graph}\,\mathit{uri}\,\{\mathit{triples}\} \parallel \mathit{process}]_{\mathit{graph} \cup \{\mathit{uri}\}}$ holds. Other rules for `from named` are similar.

Consider the operational rule for `select`. Assume that the following hold:

$$\vdash \mathit{term} \colon \mathit{type} \qquad \text{and} \qquad \vdash \texttt{select}\ \texttt{\$x} \colon \mathit{type}\ \mathit{script}$$

By the type rule for `select`, $\texttt{\$x} \colon \mathit{type} \vdash \mathit{script}$ must hold. Hence, by the Lemma 5.1, $\vdash \mathit{script}\{^{\mathit{term}}/_{\texttt{\$x}}\}$ holds.

Consider the operational rule for `where`. Assume that $\vdash \texttt{where}\,\mathit{data}\ \mathit{script} \parallel \mathit{data}$ holds. By the type rule for parallel composition, $\vdash \texttt{where}\,\mathit{data}\ \mathit{script}$ and $\vdash \mathit{data}$ must hold, and, by the type rule for `where`, $\vdash \mathit{data}$ and $\vdash \mathit{script}$ must hold. Hence, by the rule for processes, $\vdash \mathit{script} \parallel \mathit{data}$ holds.

Consider the operational rules for `union`. Assume that $\vdash \mathit{query}_0\ \texttt{union}\ \mathit{query}_1$ holds. By the rule for `union`, $\vdash \mathit{query}_0$ and $\vdash \mathit{query}_1$ hold.

Consider the operational rules for coercion of integers to decimals. Assume that $\vdash \mathit{integer} + \mathit{decimal}_1 \colon \mathit{type}$. By the rule addition and axioms for numbers, it must be the case that *type* is *xsd:decimal*. Let $\mathit{decimal}_0 = \mathit{int\_dec}(\mathit{integer})$ hence by the axiom for decimals, $\vdash \mathit{decimal}_0 \colon \mathit{xsd{:}decimal}$ and $\vdash \mathit{decimal}_1 \colon \mathit{xsd{:}decimal}$. Therefore, by the rule for addition, $\vdash \mathit{decimal}_0 + \mathit{decimal}_1 \colon \mathit{xsd{:}decimal}$. The remaining cases for expressions and predicates are similar. □

5.3. Type Safety

A type system should avoid some basic programming errors. Thus we expect that well typed scripts do not throw runtime type errors. For our type system, this claim is verified by the following theorem.

Theorem 5.3 (type safety). *If* $\vdash$ *script and* $\mathit{script} \longrightarrow \mathit{script}'$*, then* script' *does not contain an error.*

Proof. The key cases to check are those that can throw an error in the untyped language, in particular, operators in expressions, boolean predicates over expressions and from named.

Consider the case of evaluating regular expressions in boolean filters. Assume that $\vdash \texttt{regex}\,(\mathit{term}, \mathit{regex})$ holds, which holds only if $\vdash \mathit{term} \colon \mathit{xsd{:}string}$. Thus, by the axiom for strings, it must be the case that *term* is a *string*. Since the evaluation of regular expressions is decidable [2], either:

1. *regex* accepts *string* and $\texttt{regex}\,(\mathit{term}, \mathit{regex}) \longrightarrow \texttt{true}$,
2. or *regex* does not accept *string* and $\texttt{regex}\,(\mathit{term}, \mathit{regex}) \longrightarrow \texttt{false}$.

Thus `regex` cannot throw an error in a well typed script.

Consider the case of `from named`. Assume that $\vdash \texttt{from named}\ \mathit{term}\ \mathit{script}$ holds, which is only possible if $\vdash \mathit{term} \colon \mathit{xsd{:}anyURI}$ holds and $\vdash \mathit{script}$ holds. By the induction hypothesis *script* does not throw an error. Also, it must by the case that *term* is a *uri*. Hence `from named` cannot throw an error.

The remaining cases follow a similar pattern. □

In contrast to the above theorem, the following untyped script can throw an error.

```
select $x
where
    graph dbpedia:Almaty {dbpedia:Almaty  dbp:population  $x}
    regex($x, 15.*)
```

Our type system does not enjoy some properties that hold for some more strongly typed systems. For example, an erasure property says that, for a well typed program, if the type annotations are erased, then the program without annotations will behave the same as the program with annotations. However, our operational semantics relies on dynamic runtime checks based on type annotations when evaluating `select`. Therefore we cannot have a type erasure property.

Future work includes implementing an interpreter for the language based on the operational semantics, and developing a minimal type inference algorithm [46] based on the type system.

6. Discovering and Inferring Schema Information

The type system relies on a schema mapping that assigns types to URIs that can be used a properties. This schema information ensures that if the URI is used as a property in a triple, then the object of the triple will conform to the type information provided. The challenge is that schema are not guaranteed to be provided a priori. Furthermore, published data may use inconsistent schema. We discuss how to extract and infer schema information using our type system. We finish with a discussion on updating schema information in a live system.

Schema Extraction. Initially developing a schema can be a manual process requiring domain specific expertise. Fortunately, many datasets, such a DBpedia and Open Graph, publish partial schema information. The type system can be used to extract schema information embedded in data. Consider the following rules of our type system.

$$\frac{\vdash \mathit{datatype} \leq \mathcal{S}(\mathit{uri})}{\vdash \mathit{uri}\ \ \mathit{rdfs{:}range}\ \ \mathit{datatype}} \qquad \frac{\vdash \mathit{xsd{:}anyURI} \leq \mathcal{S}(\mathit{uri}_0)}{\vdash \mathit{uri}_0\ \ \mathit{rdfs{:}range}\ \ \mathit{uri}_1} \qquad \frac{\vdash \mathit{xsd{:}anyURI} \leq \mathcal{S}(\mathit{uri})}{\vdash \mathit{uri}\ \ \mathit{rdf{:}type}\ \ \mathit{owl{:}ObjectProperty}}$$

If the schema mapping $\mathcal{S}(\mathit{uri})$ is known in advance, the above rules check that the new schema is consistent with the existing schema. If, however, there exists no schema information for *uri*, we can use our type system to infer the schema for $\mathcal{S}(\mathit{uri})$. For example, from the triple *dbp:foundationDate rdfs:range xsd:dateTime*, we can infer the constraint that $\mathit{xsd{:}dateTime} \leq \mathcal{S}(\mathit{dbp{:}foundationDate})$. Constraints can be solved to find the most general unifying type that can be assigned to a property by a schema map.

Schema Inference. The type system can also be used to infer schema information when there is no schema information available for a property. Consider an example where $\mathcal{S}(\mathit{geo{:}lat})$ is unknown. We can assign a fresh variable X such that $\mathcal{S}(\mathit{geo{:}lat}) = X$, and unfold a proof tree for some data using *geo:lat* as follows.

$$\dfrac{\dfrac{\dfrac{\vdash \mathit{xsd{:}integer} \leq X}{\vdash 77 : X}}{\vdash \mathit{dbpedia{:}Almaty}\ \ \mathit{geo{:}lat}\ \ 77} \qquad \dfrac{\dfrac{\mathit{xsd{:}decimal} \leq X}{\vdash 51.1801 : X}}{\vdash \mathit{dbpedia{:}Astana}\ \ \mathit{geo{:}lat}\ \ 51.1801}}{\vdash \mathit{dbpedia{:}Almaty}\ \mathit{geo{:}lat}\ 77\ .\ \ \mathit{dbpedia{:}Astana}\ \ \mathit{geo{:}lat}\ \ 51.1801}$$

As with type inference, by solving the system of inequality constraints [51] we obtain that $\mathcal{S}(\mathit{geo{:}lat}) = \mathit{xsd{:}double}$.

A novel technique enabled by our type system is that schema information can be inferred from queries and scripts. Assume that we do not know $\mathcal{S}(\mathit{foaf{:}name})$ and consider the following query.

```
select $x where
    $y foaf:name $x .
    $y dbp:placeOfBirth dbpedia:Kazakhstan .
    regex($x, ^zhan)
```

By unfolding typing rules, we can derive the constraint $\mathcal{S}(\mathit{foaf{:}name}) \leq \mathit{xsd{:}string}$. Thus the programmer is not constrained by missing schema information.

Resolving Inconsistent Schema. However, there can be inconsistencies in the inference process. Consider the following data from DBpedia.

dbpedia:Rockmart,_Georgia dbp:subdivisionName 1872 .
dbpedia:Rockmart,_Georgia dbp:subdivisionName dbpedia:Polk_County,_Georgia .
dbp:subdivisionName rdf:type dbp:Place .

Assuming that $\mathcal{S}(\mathit{dbp{:}subdivisionName}) = Y$, we infer the constraints $\mathit{xsd{:}integer} \leq Y$ and $\mathit{xsd{:}anyURI} \leq Y$. In the subtype system presented, there is no upper bound for *xsd:integer* and *xsd:anyURI*; hence there is no solution to these constraints. This is not a deficiency of our approach but a feature of working with Linked Data, where schema cannot be fixed a priori.

There are two approaches to the above scenario, where there is an inconsistency in the inferred schema. The first is to use statistics or human intervention to resolve the inconsistency, causing some triples to be rejected. The second is to enable a more expressive subtype system with union and intersection types [3]. A more complex type system involving union types and top types is described in related work [34, 33]. Using union and intersection types requires more of OWL [32] to be lifted to our type system. In this work, we deliberately choose a simpler type system.

Monotonic Schema Update. Finally, we consider the problem that schema information may be updated while a system is live. Discovering new schema or relaxing schema information can always be done without breaking the system. However, operations that change the schema, for example change $\mathcal{S}(\textit{dbp:subdivisionName})$ from *xsd:anyURI* to *xsd:integer*, is an expensive transaction, that requires rechecking of data and scripts that explicitly involve the property. Here we consider the monotonic case and leave specifying transactional schema updates as future work.

We define an ordering over schema partial maps such that new properties may be added and datatypes may be relaxed.

Definition 6.1. $\mathcal{S}_0 \leq \mathcal{S}_1$ *if and only if, for all uri such that* $\mathcal{S}_0(\textit{uri}) = \textit{datatype}_0$*, we have that also* $\mathcal{S}_1(\textit{uri}) = \textit{datatype}_1$ *and* $\vdash \textit{datatype}_0 \leq \textit{datatype}_1$.

Notice that the above definition implies that the domain of $\mathcal{S}_0$ is a subset of the domain of $\mathcal{S}_1$, so more properties may appear in $\mathcal{S}_1$.

The following proposition verifies that, if a schema is extended monotonically with respect to Defn. 6.1, then all systems, hence all data and all scripts, remain well typed.

Proposition 6.2. *If* $\mathcal{S}_0 \leq \mathcal{S}_1$*, then if* $\Gamma \vdash \textit{system}$ *with respect to* $\mathcal{S}_0$*, then* $\Gamma \vdash \textit{system}$ *with respect to* $\mathcal{S}_1$

Proof. The proof is by structural induction.

The relevant base case is the case for uris used as properties. Assume, with respect to schema $\mathcal{S}_0$, that $\Gamma \vdash \textit{uri}\colon \textit{range}(\textit{datatype})$ holds. By the type rule for properties, this holds only if $\vdash \textit{range}(\mathcal{S}_0(\textit{uri})) \leq \textit{range}(\textit{datatype})$. Now, since $\mathcal{S}_0 \leq \mathcal{S}_1$, by definition it holds that $\vdash \mathcal{S}_0(\textit{uri}) \leq \mathcal{S}_1(\textit{uri})$; hence, by contravariance of property types, $\textit{range}(\mathcal{S}_1(\textit{uri})) \leq \textit{range}(\mathcal{S}_0(\textit{uri}))$. By Lemma 4.1, $\vdash \textit{range}(\mathcal{S}_1(\textit{uri})) \leq \textit{range}(\textit{datatype})$; from which we establish that $\Gamma \vdash \textit{uri}\colon \textit{range}(\textit{datatype})$, with respect to schema $\mathcal{S}_1$.

The relevant inductive case is the cases for triples with a specific URI as a property. Assume, with respect to $\mathcal{S}_0$, that $\Gamma \vdash \textit{term}_0\ \textit{uri}\ \textit{term}_1$. holds. By the rule for triples, this holds only if $\Gamma \vdash \textit{term}_0\colon \textit{xsd:anyURI}$ and $\Gamma \vdash \textit{term}_1\colon \mathcal{S}_0(\textit{uri})$. Since $\mathcal{S}_0 \leq \mathcal{S}_1$, by definition it holds that $\vdash \mathcal{S}_0(\textit{uri}) \leq \mathcal{S}_1(\textit{uri})$. Therefore, by Lemma 4.2, $\Gamma \vdash \textit{term}_1 : \mathcal{S}_1(\textit{uri})$ holds; from which we establish that $\Gamma \vdash \textit{term}_0\ \textit{uri}\ \textit{term}_1$ holds with respect to $\mathcal{S}_1$.

All other cases are immediate, since they do not involve the schema map. □

7. Related Work

We highlight three significant lines of related work: types and inference, domain specific scripting languages, and consuming data from the Web.

This work discusses type inference in the context of Linked Data. This is related to, but not the same as inference in the RDF Schema recommendation [11]. The formalisation of RDF Schema in [40], includes rules such as if $\textit{uri}_0\ \textit{uri}_1\ \textit{uri}_2$ and $\textit{uri}_1$ *rdfs:range* $\textit{uri}_3$ and $\textit{uri}_3$ *rdfs:subClassOf* $\textit{uri}_4$, then $\textit{uri}_2$ *rdf:type* $\textit{uri}_4$. For type inference in this paper, if we have a triple, say `$x` *foaf:name* `$y` and $\mathcal{S}(\textit{foaf:name}) = \textit{xsd:string}$, then assuming that `$y`: Y, type inference will generate the constraint $Y \leq \textit{xsd:string}$, which is similar to the above rule for RDF Schema. Despite this similarity, inference in RDF Schema and type inference are not the same thing. Related work [42], discusses a higher-order strata of types for formalising RDF Schema. We avoid the complexity of higher order types by assigning all URIs the datatype *xsd:anyURI*.

Several domain specific languages have been proposed for working with Linked Data. ActiveRDF [41] can be used as a data layer in Ruby. In ActiveRDF, as with our scripting language, queries are primitive enabling basic features such as compile time syntax checking and tight integration with control flow. Data-Fu [50] is a domain specific language that allows result of queries to be used tightly with HTTP requests, hence can be used for RESTful data APIs. NautiLOD [24] embeds queries in paths and shares the aim of our scripting language to tightly couple navigation and querying of Linked Data. Note that our scripting language is a simple core language to exhibit type inference techniques. Our type inference techniques can be extended to ActiveRDF, Data-Fu and other domain specific language extensions.

This work focusses on the Linked Data consumer, which harvests data from the Web. Harvesting data from the Web is an older problem than Linked Data, from which insight may be gleaned. An original problem for Google [12] was to crawl the Web to discover documents. To discover enough data in reasonable time, Google required a distributed and fault tolerant system. For deciding when to revisit known data efficiently, metrics are required [16]. Concerns such as these can be built into a service which handles concerns such as concurrency, fault tolerance and recrawling, while offering our scripting language as an interface. Consuming Linked Data is also related to classic work on focussed crawling [15].

8. Conclusion

Linked Data enables processes to programmatically crawl the Web of 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 Data that is tailored to a particular application. Motivated by the requirements of Web developers, 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 an application 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. To appeal to Web developers, we take care to design the syntax of the scripting language such that it resembles the SPARQL recommendations.

We introduce a simple but useful type system for our language. The type system is based on fragments of the SPARQL, RDF Schema and OWL recommendations that deal with simple data types. Types 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 the correct term appears as the object of a triple. The dynamic type checks do not impose significant restrictions on the data consumed, due to techniques for extracting and inferring schema information. Furthermore, our type system is algorithmic, so a minimal type inference algorithm can be used instead of manually adding type annotations, making programming easier.

Many datasets, including data from DBpedia, conform to the constraints imposed by our type system. Further weight is given to our design decisions by the Facebook Open Graph protocol, which demands typing at exactly the level we deliver.

Acknowledgements. We thank the reviewers and participants at WWV2013, who's comments helped us develop our results. The first and second authors were supported by a grant from the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, project number PN-II-ID-PCE-2011-3-0919.

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable Semantic Web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[2] Alfred V. Aho, Ravi Sethi Monica S. Lam, and Jeffrey D. Ullman. *Compilers: principles, techniques, & tools*. Pearson Education, 2007.

[3] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo Deliguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[4] David Beckett. The design and implementation of the Redland RDF application framework. *Computer Networks*, 39(5):577–588, 2002.

[5] Tim Berners-Lee. Information management: A proposal. 1989.

[6] Tim Berners-Lee. Linked data — design issues. 2006.

[7] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):28–37, 2001.

[8] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes second edition. Recommendation REC-xmlschema-2-20041028, W3C, 2004.

[9] Christian Bizer et al. DBpedia: A crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165, 2009.

[10] Christian Bizer and Andy Seaborne. D2rq-treating non-rdf databases as virtual rdf graphs. In *Proceedings of the 3rd International Semantic Web Conference (ISWC2004)*, page 26, 2004.

[11] Dan Brickley and Ramanathan V. Guha. RDF vocabulary description language 1.0: RDF Schema. Edited recommendation PER-rdf-schema-20140109, W3C, 2014.

[12] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1):107–117, 1998.

[13] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF Schema. In *International Semantic Web Conference*, pages 54–68, 2002.

[14] Jeremy J. Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(4):247–267, 2005.

[15] Soumen Chakrabarti, Martin Van den Berg, and Byron Dom. Focused crawling: a new approach to topic-specific Web resource discovery. *Computer Networks*, 31(11):1623–1640, 1999.

[16] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient crawling through url ordering. *Computer Networks and ISDN Systems*, 30(1):161–172, 1998.

[17] Gabriel Ciobanu and Ross Horne. A provenance tracking model for data updates. In *FOCLASA*, pages 31–44, 2012.

[18] Gabriel Ciobanu, Ross Horne, and Vladimiro Sassone. Local type checking for linked data consumers. In *Proceedings WWV 2013*, volume 123 of *EPTCS*, pages 19–33, 2013.

[19] Richard Cyganiak, David Wood, and Markus Lanthaler. Rdf 1.1 concepts and abstract syntax. Recommendation REC-rdf11-concepts-20140225, W3C, 2014.

[20] Mariangiola Dezani, Ross Horne, and Vladimiro Sassone. Tracing where and who provenance in Linked Data: a calculus. *Theoretical Computer Science*, 464:113–129, 2012.

[21] M. Dürst and M. Suignard. Internationalized resource identifiers (iris). Standards Track RFC3987, IETF, 2005.

[22] Orri Erling and Ivan Mikhailov. RDF support in the Virtuoso DBMS. In *CSSW*, pages 59–68, 2007.

[23] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, 2002.

[24] Valeria Fionda, Claudio Gutierrez, and Giuseppe Pirró. Semantic navigation on the web of data: specification of routes, web fragments and actions. In *Proceedings of the 21st international conference on World Wide Web*, pages 281–290. ACM, 2012.

[25] Paul Gearon, Alexandre Passant, and Axel Polleres. SPARQL 1.1 update. Recommendation REC-sparql11-update-20130321, W3C, 2013.

[26] Alessio Guglielmi and Tom Gundersen. Normalisation control in deep inference via atomic flows. *Logical Methods in Computer Science*, 4(1), 2008.

[27] Peter Haase, Michael Schmidt Christian Hütter, and Andreas Schwarte. The Information Workbench as a self-service platform for Linked Data applications. In *WWW 2012. Lyon, France*, 2012.

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

[29] Steve Harris and Andy Seaborne. SPARQL 1.1 query language. Recommendation REC-sparql11-query-20130321, W3C, MIT, MA, 2013.

[30] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2011.

[31] James A. Hendler. Where are all the intelligent agents? *IEEE Intelligent Systems*, 22(3):2–3, 2007.

[32] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language primer (second edition). Recommentation REC-owl2-primer-20121211, W3C, 2012.

[33] Ross Horne. *Programming Languages and Principles for Read–Write Linked Data*. PhD thesis, School of Electronics and Computer Science, University of Southampton, 2011.

[34] Ross Horne and Vladimiro Sassone. A typed model for linked data. Technical report, University of Southampton, 2011.

[35] Ross Horne and Vladimiro Sassone. A verified algebra for read-write linked data. *Science of Computer Programming*, 89(A):2–22, 2014.

[36] Ross Horne, Vladimiro Sassone, and Nicholas Gibbins. Operational semantics for sparql update. In *The Semantic Web*, volume 7185 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 2012.

[37] Ashok Malhotra, Jim Melton, Norman Walsh, and Michael Kay. Xquery 1.0 and xpath 2.0 functions and operators (second edition). Recommendation REC-xpath-functions-20101214, W3C, 2010.

[38] Alejandro Mallea, Marcelo Arenas, Aidan Hogan, and Axel Polleres. On blank nodes. In *International Semantic Web Conference (1)*, pages 421–437, 2011.

[39] Brian McBride. Jena: A Semantic Web toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.

[40] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. Simple and efficient minimal RDFS. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):220–234, 2009.

[41] Eyal Oren, Benjamin Heitmann, and Stefan Decker. Activerdf: Embedding semantic web data into object-oriented languages. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(3):191–202, 2008.

[42] Jeff Z. Pan and Ian Horrocks. RDFS (FA) and RDF MT: Two semantics for RDFS. In *The Semantic Web-ISWC 2003*, pages 30–46. Springer, 2003.

[43] A. Phillips and M. Davis. Tags for identifying languages. Best Current Practice RFC4646, IETF, 2006.

[44] Benjamin C. Pierce. *Types and programming languages*. The MIT Press, 2002.

[45] Eric Prud'hommeaux and Gavin Carothers. Rdf 1.1 turtle: Terse rdf triple language. Recommendation REC-turtle-20140225, W3C, 2014.

[46] Michael I. Schwartzbach. Type inference with inequalities. In *Proceedings of CAAP: vol 1*, TAPSOFT '91, pages 441–455. Springer, 1991.

[47] Nigel Shadbolt, Wendy Hall, and Tim Berners-Lee. The Semantic Web revisited. *IEEE intelligent systems*, 21(3):96–101, 2006.

[48] Nigel Shadbolt, Kieron O'Hara, Tim Berners-Lee, Nicholas Gibbins, Hugh Glaser, Wendy Hall, and m. c. schraefel. Linked open government data: Lessons from data.gov.uk. *IEEE Intelligent Systems*, 27(3):16–24, 2012.

[49] John F. Sowa. *Knowledge representation: logical, philosophical and computational foundations*. Brooks/Cole Publishing Co., 2000.

[50] Steffen Stadtmüller, Sebastian Speiser, Andreas Harth, and Rudi Studer. Data-fu: A language and an interpreter for interaction with read/write linked data. In *Proceedings of the 22nd World Wide Web conference*, pages 1225–1236, 2013.

[51] Jerzy Tiuryn. Subtype inequalities. In *Logic in Computer Science, 1992. LICS'92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 308–315. IEEE, 1992.