Supporting Arbitrary Custom Datatypes in RDF and SPARQL

Maxime Lefrançois(✉) and Antoine Zimmermann

École Nationale Supérieure des Mines, FAYOL-ENSMSE,
Laboratoire Hubert Curien, 42023 Saint-Étienne, France
{maxime.lefrancois,antoine.zimmermann}@emse.fr

Abstract. In the Resource Description Framework, literals are composed of a UNICODE string (the lexical form), a datatype IRI, and optionally, when the datatype IRI is `rdf:langString`, a language tag. Any IRI can take the place of a datatype IRI, but the specification only defines the precise meaning of a literal when the datatype IRI is among a predefined subset. Custom datatypes have reported use on the Web of Data, and show some advantages in representing some classical structures. Yet, their support by RDF processors is rare and implementation specific. In this paper, we first present the minimal set of functions that should be defined in order to make a custom datatype usable in query answering and reasoning. Based on this, we discuss solutions that would enable: (i) data publishers to publish the definition of arbitrary custom datatypes on the Web, and (ii) generic RDF processor or SPARQL query engine to discover custom datatypes on-the-fly, and to perform operations on them accordingly. Finally, we detail a concrete solution that targets arbitrarily complex custom datatypes, we overview its implementation in Jena and ARQ, and we report the results of an experiment on a real world DBpedia use case.

Keywords: Literals · Datatypes · RDF · Linked data

1 Introduction

The Resource Description Framework empowers the Web of Data with three kinds of entities: IRIs, blank nodes, and literals [3]. IRIs are obviously central as they allow the interlinking of datasets and serendipitous discovery of more data. Blank nodes have been the subject of several papers (a comprehensive review is found in [9]). Literals are extremely important since they are, after all, the carriers of the data that is eventually processed. In fact, we argue that IRIs are only crucial insofar as they offer a way of traversing linked data towards the discovery of literal values.

RDF defines literals as being composed of a UNICODE string and a datatype IRI[1], the latter being an *arbitrary* IRI that may refer to any datatype conforming

This work has been supported by ITEA2 project SEAS 12004.

[1] And optionally a language tag when the datatype IRI is `rdf:langString`, but for the purpose of this paper, we will simply consider literals as pairs.

H. Sack et al. (Eds.): ESWC 2016, LNCS 9678, pp. 371–386, 2016.
DOI: 10.1007/978-3-319-34129-3_23

to the definition in [3, Sect. 5]. The datatype that the IRI refers to gives meaning to the literals having that type. Indeed, by definition, a datatype defines what value the UNICODE string represents in that type.

An RDF processor that is able to distinguish the values of literals for a given datatype IRI is said to *recognise* the IRI. It is possible to program an RDF processor such that it recognises a fixed set of IRIs by implementing the associated set of specifications. Usually, the set of recognised datatypes is the set of XSD datatypes. However, even some RDF processors don't process them in a uniform way [5]. And even then, processors cannot compare literals with datatype IRIs they do not recognise. In this paper, we want to address the case of a processor that does not necessarily recognise a fixed set of IRIs but is able to determine the datatype associated with an IRI on the fly. We provide motivating use cases for this in Sect. 2.

To achieve this, we first show that an RDF processor does not necessarily need to "know" the actual datatype (which is a mathematical structure that cannot always be represented in a computer format). Instead, for some reasoning or query answering purposes, recognising a datatype amounts to using a small set of functions that can usually be provided in a computer language. We describe these functions in Sect. 3. In Sect. 4, we show several options for implementing an RDF processor that can take advantage of a computerised description of these functions such that it can recognise some new datatypes on the fly. We present our own implementation in Sect. 5. Our evaluation in Sect. 6 demonstrates that the approach does not introduce significant overhead while it makes both publishing data easier, and writing more concise queries when compared to an approach purely based on standard datatypes. Section 7 provides a critical discussion of the overall approach and our specific implementation, with an overview of future work.

2 Use Cases for on-the Fly Support of a New Datatype

This section introduces several motivating use cases for enabling on-the-fly support of custom datatypes in RDF processors and SPARQL query engines.

Sharing Energy Related Data. In the ITEA2 SEAS project that partly funds this research, industrial partners want to share energy-related data such as energy consumption and production, capacities, temperatures, and they use various custom datatypes for representing these data. Sometimes, they use different datatypes to represent similar information, such as `ex1:wattHour`, `ex2:barrelOfOilEquiv`, and `ex3:GJ` for energy quantities. RDF processors and SPARQL query engines cannot be updated for the support of each individual datatype in use. Also, it is impossible to write a SPARQL query that selects consumptions or productions that are within a given range. Instead, processors could rely on a generic mechanism for automatically retrieving sufficient information for the data to be processed, queried, and compared.

Distributed Computation. In distributed and collaborative computing, it is necessary to transfer the state of a program execution in a serialised form. A state can be shared as a combination of metadata and serialised OOP objects that can be adequately represented in RDF, with serialised objects being written as literals with a type indicating the class membership. In this situation, it could be desirable to know which executions reach the same state. This would be possible with a SPARQL query, had there been a mechanism for associating the datatype IRI to the appropriate datatype definition.

Well Known Text Literals. In the OGC standard GeoSPARQL [11], a datatype is defined for serialising geolocated region of space, such as `"LINESTRING(0 0, 1 1, 1 2, 2 2)"^^geo:wktLiteral`.[2] However, wktLiteral can also specify a coordinate reference system that differs from the default CRS84 by adding a URI at the beginning of the literal, e.g., `"<http://www.opengis.net/def/crs/EPSG/0/4326> Point(33.95 -83.38)"^^geo:wktLiteral`. There is no restriction on the URI being used at this position in the standard, so some wktLiterals may not be understood even by processors that implement the GeoSPARQL standard. Had the coordinate system been given as a datatype IRI, and assuming a mechanism as we propose to dynamically obtain the specification of the datatype, new coordinate systems could be supported as soon as they appear.

3 Requirements for on-the Fly Support of a New Datatype

In this section, we describe required functionalities to effectively recognise a datatype IRI, but first we provide preliminary definitions. For clarity, we will then restrain to the case when it is assumed that value spaces are pairwise disjoint before addressing the more general case.

3.1 Preliminaries

As mentioned in footnote 1, we only focus here on literals that do not have a language tag. Therefore, from now on, a literal will be a pair comprising a UNICODE string called the *lexical form* and an IRI called the *datatype IRI*. When we need to refer to an arbitrary IRI, we use names of the form `a`, `b`, etc. with letters from the beginning of the alphabet, while for arbitrary UNICODE string, we use names like `s`, `t`, etc. with letters from the end of the alphabet. We first recall necessary definitions from the RDF 1.1 specifications.

Definition 1 (Datatype). *A* datatype *D is a structure comprising the following components:*

[2] Subsequently, we will use `geo:` for http://www.opengis.net/ont/geosparql. Similarly, we will use usual prefixes `rdf:`, `rdfs:`, `xsd:`, and `owl:` in all examples.

- *a set $L(D)$ of UNICODE strings, called the* lexical space*;*
- *a set $V(D)$, called the* value space *of D;*
- *a mapping $L2V(D) : L(D) \to V(D)$, called the* lexical-to-value mapping*, that maps all strings in the lexical space to a value in the value space.*

To avoid paraphrasing RDF 1.1 Semantics, we only refer the most relevant definitions in [7]. In this paper, we rely heavily on the notion of *recognised IRI*, *simple D-interpretation*, and *D-entailment* (or *simple entailment recognising D*) defined in [7, Sect. 7]. We also utilise the extensions to *RDF* and *RDFS-entailment recognising D* from [7, Sects. 7 and 8]. When an RDF processor recognises an IRI identifying a datatype D_a, we say that it *supports* D_a.

3.2 Pairwise Disjoint Value Spaces

An RDF processor that supports a datatype D_a identified by an IRI `a` must be able to check two things: whether a UNICODE string belongs to the lexical space of D_a or not, and whether two literals with datatype D_a share the same value.

Well-formedness. Given a UNICODE string `s`, is the lexical form `s` well formed in D_a, i.e., Does it belong to the lexical space of D_a? Or equivalently, is literal `"s"^^a` well typed? i.e., $\mathtt{s} \in L(D_a)$.
For example, `"12.5"` is well formed in `xsd:decimal`, while `"abc"` is not (that is, `"12.5"^^xsd:decimal` is well typed, and `"abc"^^xsd:decimal` is ill-typed).

Equality. Given two UNICODE strings `s`,`t`, do `"s"^^a` and `"t"^^a` share the same value? i.e., $L2V(D_a)(\mathtt{s}) = L2V(D_a)(\mathtt{t})$.
For example, `"0.50"^^xsd:decimal` and `".5"^^xsd:decimal` share the same value.

Note that if a UNICODE string is not in the lexical space of a datatype, then it does not have a value. Hence, it would never be equal to any other literal value.

Concerning SPARQL query engines, basic graph matching only requires being able to join values, and therefore nothing more than what precedes is needed. Now, SPARQL offers an extension point related to filtering and ordering literals: SPARQL implementations may extend the XPath and SPARQL Tests operators $\{=, !=, <, >, <=, >=\}$ [6]. Apart from testing equality, SPARQL engines may need to test the ordering of literals.

Value comparison. Given two UNICODE strings `s`,`t`, is the value of `"s"^^a` lower (resp., greater) than the value of `"t"^^a`? i.e., $L2V(D_a)(\mathtt{s}) < L2V(D_a)(\mathtt{t})$ (resp., $L2V(D_a)(\mathtt{s}) > L2V(D_a)(\mathtt{t})$).

These functionalities are sufficient to check for simple D-entailment between RDF graphs, and even RDFS entailment recognising D, as long as the value spaces are infinite (and we assume here they are disjoint), as shown in [4]. The case of RDFS reasoning recognising datatypes of finite size is tricky and discussed in Sect. 7.

3.3 Overlapping Value Spaces

Now, as justified by the use cases, datatypes value spaces may overlap. Practically, we need to extend the equality and value comparison checking to different datatypes.

Cross-datatype equality. Given two datatypes D_a and D_b respectively identified by IRIs `a` and `b`, given two UNICODE strings (`s`,`t`) $\in L(D_a) \times L(D_b)$, do `"s"^^a` and `"t"^^b` share the same value? i.e., $L2V(D_a)$(`s`) $= L2V(D_b)$(`t`). For example, `"1"^^ex2:barrelOfOilEquiv` and `"6.1178632e9"^^ex3:GJ` share the same value.

Cross-datatype value comparison. Given two datatypes D_a and D_b respectively identified by IRIs `a` and `b`, given two UNICODE strings (`s`,`t`) $\in L(D_a) \times L(D_b)$, is the value of `"s"^^a` lower (resp., greater) than the value of `"t"^^b`? i.e., $L2V(D_a)$(`s`) $< L2V(D_b)$(`t`) (resp., $L2V(D_a)$(`s`) $> L2V(D_b)$(`t`)). For example, `"1"^^ex2:barrelOfOilEquiv` is bigger than `"1"^^ex3:GJ`.

These functionalities are again sufficient to check for simple D-entailment between RDF graphs. However, they may not be sufficient for RDFS entailment, even with infinite value spaces. [4] proved that if any intersection of value spaces is infinite or empty, then these functions would be sufficient to do correct and complete RDFS reasoning recognising D (see Sect. 7 for more details). Note that if these constraints are not met, it is still possible to perform sound reasoning that is complete on graphs that only use the datatype IRIs in literals rather than as subject, predicate, or object. For example, graphs can contain this type of triples: `:s :p "1"^^xsd:int`, but not this type of triples: `:p rdfs:range xsd:integer`.

4 Implementation Options

RDF processors that have to deal with a datatype IRI for which they do not have hard-coded implementation should be able to retrieve a processable version of the functions described in Sect. 3. This assumes that these functions can be computed. In general, it is not the case. For instance, a datatype could encode a FOL formula, with the value space being the set of equivalent class wrt FOL entailment. In this paper, we want to address the most general case, namely when equality, well-formedness, and comparison are all computable functions (i.e., that the associated decision problems are decidable).

For cross-datatype comparisons, our requirements suggest that it should be possible to compare literals from any datatype to literals from any other. It is not practically doable, so any solution would be partial. However, we want to provide a mechanism that makes it possible to extend to an arbitrary large finite set of supported datatypes.

Clearly, any solution must involve an agreement between both the publisher[3] and the consumer on a common mechanism for presenting and exploiting the required functionalities. In an ideal situation, a standard would exist that would

[3] Here, the publisher is the one that specifies the datatype associated with an IRI.

reduce the need for coordinating between publishers and consumers. These functions could be provided by a centralised datatype registration service, where publishers submit their datatype specification. However, such a solution is unpractical and at odd with basic web principles.

Therefore, in what follows, we focus on solutions that work on the principle that the requested functions are accessible by way of dereferencing the datatype IRI. As a matter of fact, this is precisely what RDF 1.1 Semantics suggests in Sect. 7. Therefore, in this section, all the solutions that we describe require that datatype IRIs are HTTP IRIs.

Using Processor-Specific Modules. ARQ and SESAME offer ways to register classes that implement custom SPARQL filter functions. The support of custom datatypes could be done in a similar way. Hence, the information these implementations would need to get from the datatype IRI would be a jar with the necessary class, for instance. This solution is reasonably simple, but it is implementation-specific, and the custom datatype publisher would require to write one class for each RDF processor. It also presents serious security issues, unless the RDF engine implements complicated control measures to avoid executing harmful unknown compiled code.

Using Functions Defined in a Script. Instead of using a compiled class for each implementation, this solution consists in providing the code of the required functions. The burden of interpreting the code would then fall on the designers of RDF processors. Nonetheless, a pivot language such as JavaScript, for which engine integration exists in many programming languages, would make this solution viable. Moreover, it uses the full expressivity of a programming language and hence enables the specification of arbitrary custom datatype. We chose to follow this approach for our implementation described in Sect. 5.

Using a Web Service. An alternative to provide the code directly would be to offer the same functionalities encapsulated in a web service. The drawback of this approach compared to a script is that the service needs high availability, the code cannot be cached, compiled, and optimised. Otherwise, this approach is worth investigating and we expect to do so in future work.

Declarative Vocabulary-Based Description. Using the full expressivity of a programming language to describe a custom datatype is excessive in many cases. It would hence be interesting to describe a datatype using a vocabulary, possibly inspired from the OWL 2 datatype restrictions [16]. We are currently undergoing research to define and use such a vocabulary, and this solution will be further discussed in Sect. 7.

In the context of this paper, we will focus on the script-based solution.

5 Script-Based Support of Arbitrary Custom Datatypes

We focus on a solution where the custom datatype specification is defined using a scripting language. This section defines a specific solution for this, where we

use the JavaScript language. Javascript has already been proposed as a language to implement custom funtions in SPARQL [17]. The solution we propose consists of: (i) guidelines for custom datatype publishers, including the definition of an API that the code in the JavaScript document must implement (Sect. 5.1); (ii) guidelines for RDF processors and SPARQL engines (Sect. 5.2). In a realistic setting, not all publishers will be following our guidelines, so we provide more functionalities than strictly needed for datatype support in order to make the approach robust to errors, corner cases, and missing information. This can serve as a model for other implementations, whether they are script-based, service-based, or declarative.

5.1 Guidelines for Datatype Publishers

The proposed solution requires to use an HTTP IRI `a` to identify datatype D_a, and to enable RDF processors and SPARQL engines to retrieve a JavaScript document from the datatype IRI when they look up `a` with a HTTP Accept header field that contains `application/javascript` (i.e., use content negotiation). Multiple datatypes may be defined in the same document, such as `xsd:string` and `xsd:int` that are defined in the same document at location http://www.w3.org/2001/XMLSchema. Hence the RDF processor would not know what part of the code it should execute for each datatype. Let D_a be a datatype identified by IRI `a`. We propose that the code implements a simple interface `CustomDatatypeFactory`, with a unique function `getDatatype(iri)`. When called with the string `a`, this function returns an object that holds the specification of datatype D_a, i.e., an instance of an interface `CustomDatatype`. We describe the methods in interfaces `CustomDatatypeFactory` and `CustomDatatype`, and sketch the expected behaviour of their implementations. This API and a formal set of constraints is described at http://w3id.org/lindt. All these methods take string parameters, and can generate errors as specified below.

Interface `CustomDatatype` defines a single method, `getDatatype`.

```
CustomDatatype getDatatype (iri)
```

A retrieved document contains a specification of custom datatype D_a identified by an IRI `a` if and only if `getDatatype(a)` returns an object `da` that implements interface `CustomDatatype`, and that complies with the set of constraints defined below. Such an object is called the *specification object* of datatype D_a.

Interface `CustomDatatype` defines the following set of methods.

```
String getIri()
Boolean isWellFormed (lexicalForm)
Boolean recognisesDatatype (datatypeIri)
String[] getRecognisedDatatypes ()
Boolean isEqual (lexForm1, lexForm2[, datatypeIri2])
Integer compare (lexForm1, lexForm2[, datatypeIri2])
String getNormalForm (lexicalForm1)
String importLiteral (lexicalForm, datatypeIri)
String exportLiteral (lexicalForm, datatypeIri)
```

Let `da` be the implementation of `CustomDatatype` returned by a call to `getDatatype(a)`, i.e., `da` is the specification object of D_a. First suppose that the value space of the defined datatype is disjoint with that of every other datatypes.

isWellFormed. A string `s` is in the lexical space of D_a if and only if a call to `da.isWellFormed(s)` returns boolean `true`.

isEqual. Two literals `"s"^^a` and `"t"^^a` have equal values if and only if a call to `da.isEqual(s, t)` returns boolean `true`. This method must generate an error if either `s` or `t` is not in the lexical space of D_a. Finally, this method must be reflexive, symmetric, and transitive.

getNormalForm. It is of great interest for RDF processors to be able to normalize lexical forms. For instance in the context of datatype `xsd:float`, the normal form of lexical form `42.0` is lexical form `4.2E1`. Method `getNormalForm` must return a string if the lexical form given as parameter is in the lexical space, or generate an error otherwise. Finally, this method is coherent with `da.isEqual`. Among other constraints, it is idempotent.

compare. This method must return a negative integer, zero, or a positive integer, depending on if the value of the first parameter is lower, equal, or greater than the value of the second parameter, respectively. It must generate an error if one of the parameters is not well formed, or if the literals are not comparable. Finally this method must be such that for any three well formed lexical forms `s`, `t`, and `u`,

- `da.isEqual(s,t)` $\Leftrightarrow$ `da.compare(s,t)` $= 0$;
- `da.compare(s,t)` $\times$ `da.compare(t,s)` ≤ 0;
- (`da.compare(s,t)` ≥ 0)$\wedge$(`da.compare(t,u)` ≥ 0) $\Rightarrow$ (`da.compare(s,u)` ≥ 0).

For datatypes whose value space is considered to be disjoint with that of any other datatype, the set of methods described above is sufficient to enable effective querying and RDFS reasoning recognising D, as justified in Sect. 3.2. As the value space of a datatype may intersect with that of other datatypes, interface `CustomDatatype` is completed as follows:

recognisesDatatype. Suppose the publisher of datatype `ex1:wattHour` is aware of the existence of datatype `ex3:GJ`, and knows how to compare values of `ex1:wattHour` literals with `ex3:GJ` literals, while the inverse is not true. In this case, the methods of the object that represents `ex1:wattHour` should be used to compare `ex1:wattHour` literals with `ex3:GJ` literals, and not the opposite. A datatype must recognise itself, but it does not need to recognise a datatype whose value space is disjoint with its own. Datatype D_a recognises datatype D_b identified by IRI `b` if and only if `da.recognisesDatatype(b)` returns boolean `true`.

isEqual. This method has an optional parameter which is the datatype IRI of the second literal. It must generate an error if the given IRI is not recognised. Given datatypes D_a, D_b, D_c identified by IRIs `a`, `b`, `c`, such that D_a and D_b are custom datatypes with specification objects `da` and `db`, D_a recognising D_b and D_c, D_b recognising D_c, lexical forms `s` and `t` well formed in D_a, `u` well formed in D_b, and `v` well formed in D_c, all of the following must be true:

- `da.isEqual(s,t,a)` = `da.isEqual(s,t)`
- `a`=`c` ⇒ `da.isEqual(s,u,b)` = `db.isEqual(u,s,a)`
- `da.isEqual(s,u,b)` and `db.isEqual(u,v,c)` ⇒ `da.isEqual(s,v,c)`

compare. This methods has an optional parameter which is the datatype IRI of the second literal. It must generate an error if the given IRI is not recognised. Given the same conditions as for `isEqual`, all of the following must be true:
- `da.compare(s,t,a)` = `da.compare(s,t)`;
- `da.isEqual(s,v,c)` ⇔ `da.compare(s,v,c)` = 0;
- `da.compare(s,u,b)` × `db.compare(u,s,a)` ≤ 0;
- (`da.compare(s,u,b)` ≥ 0)∧(`db.compare(u,v,c)` ≥ 0) ⇒ (`da.compare(s,v,c)` ≥ 0).

importLiteral. Given a datatype D_a identified by IRI `a` and with specification object *da*, this method takes as input a lexical form `t` and a datatype D_b identified IRI `b`, and returns a well formed lexical form `s` such that $L2V(D_a)$(`s`) = $L2V(D_b)$(`t`). If D_a does not recognise `b` or if there exists no such well formed lexical form, then the method must generate an error. Else, the following must be true:
- `da.isEqual(da.importLiteral(t,b),t,b)`

exportLiteral. Given a datatype D_a identified by IRI `a` and with specification object `da`, this method takes as input a lexical form `s` and another datatype D_b identified by IRI `b`, and returns a well formed lexical form `t` such that $L2V(D_a)$(`s`) = $L2V(D_b)$(`t`). If D_a does not recognise `b`, if *s* is not well formed, or if there exists no such well formed lexical form, then the method must generate an error. Else, the following must be true:
- `da.isEqual(s,da.exportLiteral(s,b),b)`

5.2 Guidelines for RDF/SPARQL Engines

When an RDF processor or a SPARQL query engine encounters a literal with an unknown datatype D_a identified by IRI `a`, it may attempt to retrieve the JavaScript document located at URL `a`, using an HTTP GET request with an Accept header field that contains `application/javascript`.

If it retrieves such a document, it may then call method `getDatatype(a)` to get a specification object `da` of datatype D_a. Lexical form validation or value comparisons between literals must then be equivalent to calling methods of `da`, as specified in the previous section. Finally, SPARQL query engines implement the following addition to SPARQL 1.1 Sect. 15.1 recommendation [6]: given datatypes D_a and D_b identified by IRIs `a` and `b`, D_a being a custom datatype with specification object `da` and recognising D_b, then when a SPARQL query engine compares two literals `"s"^^a` and `"t"^^b`, the ordering of these two literals must match the one given by function `da.compare(s,t,b)`.

To avoid security issues, the code may be executed in a sandbox environment without further precaution; it may undergo some static formal verifications; or it may be submitted to a trusted web service for approval. If the RDF processor or the SPARQL query engine decides not to use the datatype specification object, the datatype must be treated as an unrecognised datatype.

6 Implementation and Experiment

This section reports the implementation of these guidelines in Jena and ARQ, and the results of an experiment on a real world DBpedia use case.

6.1 Publication of a Simple Custom Datatype for Length

For illustration purposes, we introduce a custom datatype to represent lengths. This datatype is identified by IRI http://w3id.org/lindt/v1/custom_datatypes#length, abbreviated as `cdt:length`. Its lexical space is the concatenation of the lexical form of an `xsd:double`, an optional space, and a unit that can be either a metric length unit, or an imperial length unit, in abbreviated form or as full words, in singular or plural form. The value space corresponds to the set of lengths, as defined by the International Systems of Quantities, i.e., any quantity with dimension distance. The lexical-to-value mapping maps lexical forms with units in the metric system to their corresponding length according to the International Systems of Quantities, while the forms with an imperial unit are mapped to their equivalent length according to the International yard and pound agreement. For example, all literals below are well typed and share the same value.

```
"1 mile"^^cdt:length                "1.609344 km"^^cdt:length
"5280 ft"^^cdt:length               "1609.344 metre"^^cdt:length
"63360 in."^^cdt:length             "1.609344E+6 mm"^^cdt:length
```

We published a JavaScript implementation of the specification of `cdt:length`, following the guidelines of Sect. 5.1. We further followed best practices for data on the Web, and serve the most appropriate document using content negotiation:

- if the HTTP header option Accept contains `text/html` or `application/xhtml+xml`, then a HTML document containing a human readable description of datatype Length is served;
- if it contains `text/turtle`, then a short RDF description of datatype Length is served;
- if it contains `application/javascript`, then a JavaScript document that contains the actual specification of custom datatype Length is served. This is equivalent to calling http://w3id.org/lindt/v1/custom_datatypes.js#length.

6.2 Implementation in Jena and ARQ

We implemented the support for on-the-fly custom datatype recognition in both the Jena RDF processor, and the ARQ SPARQL engine.[4] It follows guidelines from Sect. 5.2, but for now it only supports custom datatypes whose value space is disjoint from that of any other datatype.

[4] https://github.com/maximelefrancois86/jena.

A new attribute `enableDiscoveryOfCustomDatatypes` has been added to class `JenaParameters`, and package `com.hp.hpl.jena.datatypes` has been slightly modified as follows: If Jena parameter `JenaParameters.enableDiscoveryOfCustomDatatypes` is set to true, then:

1. When method `getSafeTypeByName` from class `TypeMapper` is called with an unknown datatype IRI `a`, it calls static method `getCustomDatatype` of a new Jena class `CustomDatatype` for a new instance of `RDFDatatype`.
2. This method first makes an HTTP call to `a`, with an `Accept: application/javascript` HTTP header field, and follows redirects. If a JavaScript document is retrieved, its code is evaluated in the default JavaScript script engine (Oracle Nashorn in Java 1.8). An instance of interface `CustomDatatypeFactory` (see Sect. 5.1) is then compiled. Its method `getDatatype` is called and an instance of interface `CustomDatatype` is compiled. This instance is wrapped in an instance of Jena class `CustomDatatype`, and sent back to the `TypeMapper`.
3. Most methods of Jena class `CustomDatatype` wrap calls to the compiled instance of interface `CustomDatatype`.

In ARQ, the main modification concerns class `NodeValue` in package `com.hp.hpl.jena.sparql.expr`, which is used for the SPARQL operators equal and less-or-equal, and for the ORDER BY clause. When comparing node values, if their datatype is an instance of `CustomDatatype`, then calls to the compiled instance of interface `CustomDatatype` are made. A few other minor modifications have also been required:

– a new instance of `ValueSpaceClassification` has been added;
– in package `com.hp.hpl.jena.sparql.expr.nodevalue`, class `NodeValueCustom` has been added, and class `NodeValueVisitor` has been modified.

6.3 Experiment

In this section we present the results of evaluating the proposed protocol, and report on the performances on loading and querying three datasets based on DBpedia but with different approaches for representing lengths. All the details, resources, and instructions that enable the reproduction of this experiment can be found at URL http://w3id.org/lindt.

Datasets. We base our datasets on the DBpedia 2014 English specific mapping-based properties dataset, which contains 819,764 triples with 21 custom datatypes among those DBpedia defines. From this, we extracted the 223,768 triples that describe lengths,[5] i.e., those with the following datatypes:

– `http://dbpedia.org/datatype/millimetre`
– `http://dbpedia.org/datatype/centimetre`

[5] This dataset is available at http://wiki.dbpedia.org/Downloads#3.

– `http://dbpedia.org/datatype/metre`
– `http://dbpedia.org/datatype/kilometre`

For instance, the following triple represents the length of the Bathyscaphe Trieste submarine.

```
dbpedia:Bathyscaphe_Trieste
  <http://dbpedia.org/ontology/MeanOfTransportation/length>
      "17983.2"^^dbpdt:millimetre .
```

We call this dataset DBPEDIA. From this dataset, we generated the dataset CUSTOM by making all literals use the same datatype Length. For example, the same fact is represented as follows:

```
dbpedia:Bathyscaphe_Trieste
  <http://dbpedia.org/ontology/MeanOfTransportation/length>
      "17983.2 mm"^^cdt:length .
```

Finally, we generated a third dataset, QUDT, which used the QUDT [8] ontology to model the same facts. This is among the alternative choices for representing physical measures that only relies on standard datatypes and encode the relationship between the value and the unit in a graph, using an ontology of quantities.[6] As an example, the length of the Bathyscaphe Trieste submarine may be modelled as follows with the QUDT ontology:

```
dbpedia:Bathyscaphe_Trieste
  <http://dbpedia.org/ontology/MeanOfTransportation/length>
  [ qudt:quantityValue
     [ qudt:numericValue "17983.2"^^xsd:double ;
       qudt:unit qudt-unit:millimetre ] ] .
```

Finally, each dataset has been derived in four datasets, that contain the first 100 %, 50 %, 25 %, and 12.5 % of the original dataset.

Queries. Besides evaluating the loading time of each dataset, we evaluated the querying time of the following simple query: *Return the 100 triples that concern the biggest lengths that are lower than* 5 m*, order the results according to the descending order of the length.* Depending on the dataset, this query writes differently. Let us just note the conciseness of the query for dataset CUSTOM:

```
PREFIX cdt: <http://w3id.org/lindt/v1/custom_datatypes#>
SELECT ?x ?prop ?length WHERE {
  ?x ?prop ?length .
  FILTER( datatype(?length) = cdt:length
         && ?length < "5m"^^cdt:length )
}
ORDER BY DESC( ?length )
LIMIT 100
```

[6] Note that using complex graph structures for representing physical quantities would solve the problem of datatype support, but it displaces the problem to the level of ontologies, as there exists many for describing measurements (in chronological order, UCUM in OWL [2], MUO [13], QUDV [1], OM [14], QUDT [8]).

Table 1. Average and standard deviation of loading or querying time of datasets (in ms).

	DBPEDIA	CUSTOM, cold start	CUSTOM, hot start	QUDT
12.5%	155(±10)	915(±53)	469(±85)	381(±82)
25%	350(±12)	1434(±46)	980(±36)	892(±13)
50%	731(±16)	2498(±57)	2013(±46)	1829(±33)
100%	1640(±57)	4659(±155)	4173(±128)	3796(±68)

(a) Average and standard deviation of loading time of datasets (in ms).

	DBPEDIA	CUSTOM	QUDT
12.5%	416(±38)	195(±12)	267(±66)
25%	833(±16)	391(±13)	532(±26)
50%	2371(±42)	784(±26)	1079(±118)
100%	4158(±256)	1593(±98)	1143(±73)

(b) Average and standard deviation of querying time of datasets (in ms).

	CUSTOM	QUDT
12.5%	35(±1)	12(±47)
25%	78(±4)	213(±13)
50%	154(±3)	411(±26)
100%	402(±35)	329(±84)

(c) Average and standard deviation of evaluation of query HEIGHT (in ms).

Experiment Protocol and Results. For a given dataset, the experiment consists in repeating 100 times: (i) resetting the `TypeMapper` instance, (ii) loading the dataset, and (iii) querying the dataset and iterating through all the results. Duration of steps ii and iii were measured, and we report below the average duration and the standard deviation of these durations. We led twice the experiment for datasets CUSTOM: once with "cold start", where the custom datatype is discovered and loaded during step (ii), and once with "hot start", where the custom datatype is manually loaded before step (ii). This difference only affects loading times. The experiments were run on a server with a 64 bits Intel Xeon® CPU E5-1603 v3 processor with 4 cores at 2.80 GHz, it has 32 GB DDR3 RAM and is running Ubuntu 14.04 LTS.

Table 1a and b report loading and querying times, respectively. Loading times of datasets CUSTOM are very close to those of datasets QUDT, with on average 468 ms penalty for discovering and loading the custom datatype in the case of cold start. On the other hand, datasets CUSTOM have the best performance regarding querying time.

- Querying time of datasets CUSTOM is between 33 % and 47 % that of datasets DBPEDIA. This can be explained by the fact that the query for dataset DBPEDIA hides actually 4 queries: one for each datatype that represents a length. We believe this difference would grow if dbpedia was using more custom datatypes to represent lengths.
- Querying datasets CUSTOM is also slightly faster than querying datasets QUDT, except for 100 % of triples. Yet, the query for datasets QUDT actually has an anchor IRI to start with, whereas the base of the query for dataset CUSTOM has none.

To evaluate the impact of having an IRI anchor, we derived a second SPARQL query, HEIGHT, by fixing the predicate URI to http://dbpedia.org/ontology/Person/height. Table 1c reports querying times of this query on datasets CUSTOM and QUDT. Fixing this IRI has a greater impact on querying time of datasets CUSTOM than on datasets QUDT, because this query already had anchor IRIs. These results show that custom datatypes have low impact on loading and querying time, while increasing the genericity of this solution.

7 Discussion

Our proposal is only partially addressing the problem of dealing with custom datatypes in a generic way. It also has shortcomings that we discuss here, with possible ways to avoid them. We first discuss the drawbacks of our implementation. We then describe possible extensions of our work to more completely support custom datatypes processing, and emphasise the relationship with OWL 2 custom datatype definitions. We then examine how well our proposal enables D-entailment reasoning.

Drawbacks. Executing code found online presents a potential security threat. However, the `CustomDatatypeFactory` indirection already represents a kind of protection. Actual custom datatype specification objects may be stored as private members of function `getDatatype(iri)`. Then, the next loaded executable code cannot modify its definition. The RDF processor must be sure that the next loaded executable code could not modify previously loaded executable codes, and that it calls the right `getDatatype(iri)` method to get the definition of D: the one that has been retrieved at its IRI. The RDF processor could also execute the code in a sandbox environment, or perhaps apply static analysis to identify harmful code.

Besides, the use of a full-fledged programming language is a bit of an overkill for simple cases such as restricting existing datatypes. We discuss the case of a declarative description of the datatype.

Extending Datatype Description. In several cases, a simple declarative description of a datatype is sufficient. As a matter of fact, OWL 2 already provides means to define datatypes that restrict some of the W3C-standard datatypes using constraining facets `xsd:length`, `xsd:minLength`, `xsd:maxLength`, and `xsd:pattern`. Similarly, we could provide a declarative description of custom datatypes based on other existing ones. Examples of what this vocabulary could represent include:

- A datatype for lengths could be derived from a datatype for measured quantities in all units, as proposed by the Unified Code for Units of Measure [15];
- Describing composite datatypes, formed from the combination of lexical separators and multiple standard literals (e.g., vectors of `xsd:integer`). An RDF processor could then use its support of the derived datatypes to support the composite datatype;

– XSD type definition components and facets [12] could be provided declaratively;
– Direct relationships between datatypes could be used, such as disjointness or subtyping. From an operational point of view, such relations could speed up decisions but would have complicated consequences on reasoning.

Such a vocabulary would favour the interlinking of datatypes.

Reasoning with Custom Datatypes. The expressiveness of RDF with custom datatypes is unlimited. To make this clear, consider a datatype where the lexical space is the set of Turtle documents, and the value space contains the equivalent classes of RDF graphs according to the OWL 2 RDF-based semantics entailment regime (a.k.a OWL 2 Full). The lexical-to-value mapping is the obvious mapping from the documents to their class of equivalent OWL Full ontologies. Equivalence in OWL 2 Full is known to be undecidable [10] and therefore, D-entailment when D contains such a datatype is undecidable. Therefore, reasoning with datatypes is generally undecidable. In fact, even simple datatypes can impact D-entailment reasoning deeply, as witnessed by the following example:

```
rdfs:Resource rdfs:subClassOf xsd:nonNegativeInteger,
    xsd:nonPositiveInteger .
```

These two triples are inconsistent in RDFS recognising {`xsd:nonNegativeInteger`, `xsd:nonPositiveInteger`} but reasoners implemented in Jena, Corese, and Sesame are unable to detect it. However, as noted in Sect. 3, under certain constraints on datatype value spaces or input graphs, our solution allow correct and complete reasoning for RDFS recognising D. Even in a more general case, reasoning is at least sound.

8 Conclusions

Custom datatypes are currently frown upon because they do not facilitate interoperability. If custom datatypes could be more easily supported generically, it would ease the publication of some domain-specific datasets which otherwise are difficult to represent with standard datatypes. We defined requirements for supporting arbitrary datatypes in reasoning and querying and proposed a concrete solution that requires that the designers of new datatypes follow guidelines that are in line with Linked Data principles. Assuming these guidelines are followed, RDF processors and SPARQL engines can effectively take advantage of custom datatypes on-the-fly, modulo a little overhead in implementing support for our proposal. We empirically demonstrated that performance is not much impacted, compared to a standard implementation. In some cases, relying on custom datatypes leads to better results than restructuring the data to only use standard ones. Arguably, in the use cases we identified, custom datatypes make data publishing more flexible, intuitive, and efficient. Nonetheless, we are conscious of some of the shortcomings of our approach and are investigating other directions for concretely implementing the requirements, based on a linked

datatype vocabulary and web services. Finally, we want to investigate more deeply real needs from data publishers in exposing their own datatypes to the open Web.

References

1. Quantities, Units, Dimensions, Values (QUDV). SysML 1.2 Revision Task Force Working draft, Object Management Group, 30 October 2009
2. Bermudez, L.: The unified code for units of measure in OWL. OWL Ontology (2006). https://marinemetadata.org/files/mmi/ontologies/ucum,accessed12/04/2016
3. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation, 25 February 2014
4. de Bruijn, J., Heymans, S.: Logical foundations of RDF(S) with datatypes. J. Artif. Intell. Res. **38**, 535–568 (2010)
5. Emmons, I., Collier, S., Garlapati, M., Dean, M.: RDF literal data types in practice. In: Proceedings of the 7th International Workshop on Scalable Semantic Web Knowledge Base Systems, vol. 1 (2011)
6. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language - W3C Working Draft 5. W3C Working Draft, W3C, 5 January 2012
7. Hayes, P., Patel-Schneider, P.F.: RDF 1.1 Semantics, W3C Recommendation 25. W3C Recommendation, W3C, 25 February 2014
8. Hodgson, R., Keller, P.J., Hodges, J., Spivak, J.: QUDT - Quantities, Units. Dimensions and Data Types Ontologies. Technical report, NASA (2014)
9. Hogan, A., Arenas, M., Mallea, A., Polleres, A.: Everything you always wanted to know about blank nodes. J. Web Semant. **27**, 42–69 (2014)
10. Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language Profiles (2nd edn.). W3C Recommendation, W3C, 11 December 2012
11. Perry, M., Herring, J.: OGC GeoSPARQL - A Geographic Query Language for RDF Data. Ogc implementation standard, Open Geospatial Consortium, 10 September 2012
12. Peterson, D., Gao, S., Malhotra, A., Sperberg-McQueen, C.M., Thompson, H.S.: W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, W3C Recommendation, W3C, 5 April 2012
13. Polo, L., Berrueta, D.: MUO - Measurement Units Ontology, Working Draft DD April 2008. Working draft, Fundación CTIC (2008)
14. Rijgersberg, H., van Assem, M., Top, J.L.: Ontology of units of measure and related concepts. Semant. Web J. **4**(1), 3–13 (2013)
15. Shadow, G., McDonald, C.J.: The Unified Code for Units of Measure. Technical report, Regenstrief Institute Inc., 22 October 2013
16. W3C OWLWorking Group: OWL 2 Web Ontology Language Document Overview (Second Edition), W3C Recommendation 11 December 2012. Technical report, W3C (2012)
17. Williams, G.: Extensible SPARQL functions with embedded javascript. In: Proceedings of the Workshop on Scripting for the Semantic Web (2007)