Datatypes for Lists and Maps in RDF Literals

Olaf Hartig[1,2⋆], Gregory Todd Williams[1], Michael Schmidt[1], Ora Lassila[1], Carlos Manuel Lopez Enriquez[1], and Bryan Thompson[1]

[1] Amazon Neptune Team, Amazon Web Services, Seattle, WA, USA
[2] Linköping University, Linköping, Sweden

Abstract. We present an approach to represent composite values (lists and maps, in particular) as literals in RDF data, and to extend SPARQL with features related to such literals. These extensions include an aggregation function to produce these composite values, functions to operate on these composite values in expressions, and a new operator to unfold such composite values into their individual components. As resources related to the approach, we provide two complete open source implementations, a formal specification, and a comprehensive test suite.

Keywords: Composite Datatypes, SPARQL Extension, JSON in RDF

1 Introduction

Composite datatypes (CDTs) enable the representation of complex, possibly nested data structures such as lists, maps, and sets. A popular mechanism to represent such complex values is JSON, which nowadays is commonly supported as a built-in datatype in database systems, including in relational systems such as MySQL and PostgreSQL. Similarly, in the graph database world, Property Graph query languages such as Gremlin and openCypher include support for CDTs such as lists, maps, and paths. Novel Web-related query languages such as GraphQL also focus on composite JSON-like structures.

In all these cases, CDTs are included as first-class citizens within the (storage or runtime) data model, and query languages offer built-in support for constructing, accessing, and manipulating composite values. Based on these observations, we argue that CDT support in RDF (and its query language, SPARQL) lags behind the state of the art: instead of supporting them as built-in types, RDF introduces so-called containers and collections [1], which allow users to model composite values through dedicated vocabulary *on top* of the core data model.

Figures 1(a) and 1(b) illustrate these options using an example with a list of three keynote speakers, Amy, Bob, and Cal, for some conference that is identified by the IRI `:CTConf`. Figure 1(a) utilizes an `rdf:List` collection, which models the list using two pointers, one to the first element (the triples with predicate `rdf:first`) and one to the tail of the list (predicate `rdf:rest`). The alternative in Figure 1(b) uses an `rdf:Seq` container, in which the list members are enumerated using a sequence of so-called membership predicates `rdf:_1`, `rdf:_2`, `rdf:_3`.

⋆ Olaf Hartig is appointed both as an Amazon Scholar and as a Senior Associate Professor at Linköping University. This paper describes work performed at Amazon.

```
:CTConf :keynoteSpeakers :List0 .
:List0 rdf:type rdf:List .
:List0 rdf:first <http://ex.com/Amy> .
:List0 rdf:rest :List1 .
:List1 rdf:first <http://ex.com/Bob> .
:List1 rdf:rest :List2 .
:List2 rdf:first <http://ex.com/Cal> .
:List2 rdf:rest rdf:nil .
```

(a) Representation as `rdf:List`

```
:CTConf :keynoteSpeakers :Seq0 .
:Seq0 rdf:type rdf:Seq .
:Seq0 rdf:_1 <http://ex.com/Amy> .
:Seq0 rdf:_2 <http://ex.com/Bob> .
:Seq0 rdf:_3 <http://ex.com/Cal> .
```

(b) Representation as `rdf:Seq`

```
:CTConf :keynoteSpeakers
 "[<http://ex.com/Amy>, <http://ex.com/Bob>, <http://ex.com/Cal>]"^^cdt:List.
```

(c) Our proposal: representation as compact, self-contained RDF literal

Fig. 1. Example of different options to represent a list in RDF.

Modeling composite values as structures within the data itself—instead of representing them as compact, self-contained objects as proposed in this paper—comes with several drawbacks. First, representing composite values becomes verbose and bloats up the storage footprint, especially when it comes to large containers and collections. Second, extracting information from such composite values using SPARQL is tricky; for instance, in the (common) case where the size of an `rdf:List` is not known upfront, returning an ordered enumeration of elements using SPARQL requires a complex query containing a mix of property paths, grouping, and counting [2]. Third, the manipulation of composite values using SPARQL is complex; for instance, writing a query that inserts an element into (a given position) of an `rdf:List` or an `rdf:Seq` is hard to achieve using SPARQL update statements, if possible at all. Ultimately, all these aspects impact the usability and performance of handling composite values in RDF [2].

Our proposal is to introduce composite type literals in RDF—as illustrated in Figure 1(c) for the running example—and to support them in SPARQL as first-class citizens. To facilitate the latter we propose language extensions for SPARQL to construct, access, and manipulate composite values at query and update time. By building upon the RDF literal mechanism, this approach is fully compatible with RDF, which means that it enables storage and retrieval of composite values as "black box" entities in existing triple stores, without modifications. Of course, systems that support the approach may leverage dedicated data structures to efficiently implement our proposed language extensions for SPARQL. The remainder of this short paper outlines the approach in more detail and describes the resources that we provide for the approach, which include a formal specification, tests suites, and two open source implementations.

2 Approach

The basis of the approach is to capture lists and maps as RDF literals with the datatype IRIs `cdt:List` and `cdt:Map`, respectively. The components of such a composite value may be RDF terms, including literals representing other composite values. The lexical form (i.e., the string representation) of such a `cdt:List` or `cdt:Map` literal contains the components of the composite value serialized in a format that is based on the RDF Turtle format [3]. For instance, the literal in the object position of the triple in Figure 1(c) represents a list of three IRIs.

An example of two lists that contain literals are given in the following triples. This example illustrates that the Turtle shorthand notation for specific types of literals can be used inside the lexical forms of `cdt:List` (and `cdt:Map`) literals.

```
:s :p1 "[1, 2, 'hello', <http://example.org/>, [1,2,3], 2.5]"^^cdt:List .
:s :p2 "['1999-08-16'^^<http://www.w3.org/2001/XMLSchema#date>,4]"^^cdt:List .
```

Maps (i.e., collections of key-value pairs) are represented by `cdt:Map` literals as in the object position of the following triple.

```
:s :p "{ 'name': 'Warsaw'@en, 1: <http://example.org/>, 9: [1,2] }"^^cdt:Map .
```

By adopting the Turtle shorthand notation, the syntax of our approach is designed such that the lexical form of `cdt:Map` literals encompasses the grammar of JSON objects, including the possibility of nested structures.

Given such literals, we extend SPARQL in the following three ways with functionality related to the types of composite values that these literals capture.

First, we introduce various functions for such literals that can be used in expressions (as used in BIND clauses, FILTER clauses, ORDER BY clauses, HAVING clauses, and SELECT clauses). As an example, consider the following SPARQL query (prefix declarations omitted) that uses two such functions in a BIND clause; the function denoted by the IRI `cdt:concat` concatenates two lists, returning the resulting list as a `cdt:List` literal again, and the other function (`cdt:size`) returns the cardinality of the resulting list. When executing this query over the first example data above (the example with the two lists), the value produced for the `?combinedLength` variable would be 8.

```
SELECT * WHERE {  :s :p1 ?l1 .
                  :s :p2 ?l2 .
                  BIND( cdt:size(cdt:concat(?l1,?l2)) AS ?combinedLength )  }
```

Further functions for `cdt:List` literals that we introduce are `cdt:contains`, `cdt:get` `cdt:head`, `cdt:reverse`, `cdt:subseq`, and `cdt:tail`; where `cdt:get` and `cdt:size` are also defined for `cdt:Map` literals, in addition to `cdt:containsKey`, `cdt:get`, `cdt:keys`, `cdt:merge`, `cdt:put`, `cdt:remove`, and `cdt:size`. Additionally, we introduce constructor functions for these literals and define corresponding extensions to the SPARQL comparison operators such as `=` and `<`.

As our second extension to SPARQL, we introduce a new operator called UNFOLD that splits composite values into their individual components and, then, assigns these components separately to a new query variable. The following query illustrates how this operator can be used to extract all elements from all lists represented by the objects of triples that match a given triple pattern. When executing this query over the first example data above (again, the one with the two lists), the query result consists of eight solutions: six for the six elements of the list in the first triple of the example data and another two for the two elements of the list in the second triple.

```
SELECT ?element ?list WHERE {  :s ?p ?list .
                               UNFOLD( ?list AS ?element )  }
```

In addition to the one-variable version of UNFOLD as demonstrated above, we also define a two-variables version, which can be used for extracting the key-value pairs from `cdt:Map` literals.

Our third extension to SPARQL is an aggregation function called FOLD that produces composite values (as `cdt:List` or `cdt:Map` literals) for groups of solution mappings. The following query illustrates how this function can be used to create lists of persons that have the same name.

```
SELECT ?name (FOLD(?person) AS ?list) WHERE { ?person rdf:type foaf:Person .
                                              ?person foaf:name ?name . }
GROUP BY ?name
```

3 Resources

We have defined our approach in a specification[3] that we aim to submit to the SPARQL-DEV Community Group[4] at the W3C to be considered for standardization. Currently, our specification is maintained in a public Github repository.[5]

The specification defines the two datatypes in terms of their respective value space, lexical space, and lexical-to-value mapping, as required by the standard mechanism to extend RDF with custom datatypes. Additionally, the specification defines the corresponding extensions to SPARQL, including:

- extensions to existing SPARQL comparison operators such as `=` and `<` that define these operators for pairs of `cdt:List` and pairs of `cdt:Map` literals,
- new functions to construct and to access such literals in expressions,
- ordering behavior for such literals in ORDER BY clauses, and
- new operators to fold and unfold the represented lists and maps in queries.

In addition to the specification document, we provide a comprehensive collection of test suites in the aforementioned Github repository. These tests cover all relevant aspects and special cases of all the extensions to SPARQL listed above and are specified in RDF using the framework[6] that was defined by the W3C RDF Data Access Working Group[7] and is now maintained by the RDF Test Curation Community Group[8] at the W3C. Since the test harnesses of many RDF and SPARQL systems are built on this framework, our test suites can readily be used when implementing support for our proposal in such systems.

[3] https://w3id.org/awslabs/neptune/SPARQL-CDTs/spec/latest.html
[4] https://www.w3.org/community/sparql-dev/
[5] https://github.com/awslabs/SPARQL-CDTs
[6] https://www.w3.org/2001/sw/DataAccess/tests/README.html
[7] https://www.w3.org/2001/sw/DataAccess/homepage-20080115
[8] https://www.w3.org/community/rdf-tests/

Besides the specification and the tests, as resources to facilitate implementations, we have also already created *two complete, Open Source implementations* of the proposal. That is, we have integrated support for the approach directly into the RDF programming frameworks Apache Jena[9] (Java) and Attean[10] (Perl).

4 Future Work

As our future work, in addition to the aforementioned plans to submit our proposal to the W3C (and also to other RDF triple store vendors), we are planning to study the performance that can be achieved with the proposed approach in our implementations. Moreover, we aim to extend the approach with options to explicitly capture typing constraints regarding elements of lists or maps, and we consider adopting the approach to also capture paths as a possible query result.

References

1. D. Brickley and R. Guha. RDF Schema 1.1. W3C Recommendation, Online at `https://www.w3.org/TR/rdf11-schema/`, Feb. 2014.
2. E. Daga, A. Meroño-Peñuela, and E. Motta. Sequential Linked Data: The State of Affairs. *Semantic Web*, 12(6):927–958, 2021.
3. E. Prud'hommeaux and G. Carothers. RDF 1.1 Turtle. W3C Recommendation, Online at `https://www.w3.org/TR/turtle/`, Feb. 2014.

[9] `https://jena.apache.org/` — Our implementation is currently in the following fork of the official Jena repository for which we are planning to request a merge. `https://github.com/hartig/jena/tree/UnfoldAndFoldWithCompositeValues`

[10] `https://github.com/kasei/attean` — Our implementation is in the following branch of Attean, ready to be merged after discussion with the Attean community. `https://github.com/kasei/attean/tree/mutli-value-exprs`