D-REPR: A Language for Describing and Mapping Diversely-Structured Data Sources to RDF

Binh Vu
USC Information Sciences Institute
Marina del Rey, CA
binhvu@isi.edu

Jay Pujara
USC Information Sciences Institute
Marina del Rey, CA
jpujara@isi.edu

Craig A. Knoblock
USC Information Sciences Institute
Marina del Rey, CA
knoblock@isi.edu

ABSTRACT

Publishing data sources to knowledge graphs is a complicated and laborious process as data sources are often heterogeneous, hierarchical and interlinked. As an example, food price datasets may contain product prices of various units at different markets and times, and different providers can have many choices of formats such as CSV, JSON or spreadsheet. Beyond data formats, these datasets may have differing layout, where one dataset may be organized as a row-based table or relational table (prices are in one column), while another may use a matrix table (prices are in one matrix). To address these problems, we present a novel data description language for mapping datasets to RDF. In particular, our language supports specifying the locations of source attributes in the sources, mapping of the attributes to ontologies, and simple rules to join the data of these attributes to output final RDF triples. Unlike existing approaches, our language is not restricted to specific data layouts such as the Nested Relational Model, or to specific data formats, such as spreadsheet. Our broad data description language presents a format-independent solution, allowing interlinking among multiple heterogeneous sources and representing many diverse data structures that existing tools are unable to handle.

CCS CONCEPTS

• **Information systems → Information integration**.

KEYWORDS

RDF mapping, Linked Data, Knowledge Graph

ACM Reference Format:
Binh Vu, Jay Pujara, and Craig A. Knoblock. 2019. D-REPR: A Language for Describing and Mapping Diversely-Structured Data Sources to RDF. In *Proceedings of the 10th International Conference on Knowledge Capture (K-CAP '19), November 19–21, 2019, Marina Del Rey, CA, USA.* ACM, Marina del Rey, CA, USA, 8 pages. https://doi.org/10.1145/3360901.3364449

1 INTRODUCTION

The WWW provides an enormous number of valuable datasets, but the variety and complexity of these sources creates a barrier to their widespread use. The recent growth of knowledge graphs as well as the Linked Open Data movement have sought to overcome these barriers by providing the platforms and common semantic conventions to allow data to be semantically defined, interlinked between sources, and used by all. However, mapping data to RDF to publish into knowledge graphs demands expertise and significant effort because datasets are in different formats (e.g., spreadsheets, JSON, NetCDF) and different layouts (e.g., row-based, matrix, hierarchical layouts). In this paper, we introduce D-REPR (Dataset Representation), a rich mapping language that addresses these two challenges while extending to many datasets that existing approaches (e.g., RML [2], XLWrap [4]) cannot easily model.

Many languages and tools have been proposed to make mapping data easier and less laborious. R2RML and its extensions [2, 6, 8] can map datasets with heterogeneous formats such as CSV, JSON, and XML. However, the mapped datasets have to be in a row-based layout, compliant with the Nested Relational Model (NRM). For datasets that are not in NRM layout (e.g., Figure 1), format-specific solutions have been proposed. For instance, XLWrap is a powerful tool to map spreadsheets to RDF, but cannot deal with other data formats. No single data mapping tool can handle the diverse set of data format and data layout choices currently in use.

Having one general RDF mapping language for many types of data sources is desirable as it currently requires much time and effort for end-users to learn different tools for different kinds of sources. However, designing such a language is challenging for several reasons. First, data sources with different formats have different protocols to access and refer to data values. Even with the same data format, data sources can be represented in different layouts. For example, a life expectancy dataset in CSV format can organize its data in matrix or in row-based layout (Figure 1a and 1b), or a museum dataset in JSON format can organize its data in a custom layout or NRM layout (Figure 1c and 1d).

To address these issues, we present a novel data description language for converting data to RDF. In our approach, users define locations of source attributes in the dataset using JSONPath, then create a semantic model that specifies semantic types of attributes by mapping each attribute to a property of an ontology class, and semantic relations between them. Finally, users define simple rules to join attributes to create tables containing instances of ontology classes. The language also has built-in support for data cleaning using transformation functions.

The contribution of this paper is a novel data description language that makes it easy for mapping format and layout heterogeneous datasets to RDF. Our approach is capable of mapping a wide variety of data sources and goes beyond the set of sources that existing mapping languages support. Furthermore, the language is extensible to new formats and new data layouts.

K-CAP '19, November 19–21, 2019, Marina Del Rey, CA, USA

ACM ISBN 978-1-4503-7008-0/19/11…$15.00
https://doi.org/10.1145/3360901.3364449

a. life table (csv format, original layout)

		2016		2015	
		Male	Female	Male	Female
ex - expectation of life at age x	<1 year	57.7	59.6	57.3	59.1
	1-4 years	60.6	62.1	60.3	61.7
	5-9 years	58.7	60.2	58.4	59.9
	10-14 years	54.4	56	54.2	55.6
	15-19 years	49.8	51.3	49.6	51

b. life table (csv format, row-based layout)

Indicator	Age group	Year	Gender	Value
ex - expectation	<1 year	2016	Male	57.7
ex - expectation	<1 year	2016	Female	59.6
ex - expectation	<1 year	2015	Male	57.3
ex - expectation	<1 year	2015	Female	59.1
ex - expectation	<1 year	2014	Male	56.2
ex - expectation	<1 year	2014	Female	58.4

c. museum dataset (json format, original layout)

```
{
  "url": "https://npg.si.edu/object/npg_NPG.70.36",
  "title": "Apollo 11 Crew",
  "sitter": [
    "Neil Alden Armstrong",
    "Michael Collins"
  ],
  "sitter_born_died_date": [
    "5 Aug 1930 - 25 Aug 2012",
    "born 20 Jan 1930"
  ]
}
```

d. museum dataset (json format, NRM layout)

```
{
  "url": "https://npg.si.edu/object/npg_NPG.70.36",
  "title": "Apollo 11 Crew",
  "sitters": [
    {
      "name": "Neil Alden Armstrong",
      "born_died_date": "5 Aug 1930 - 25 Aug 2012"
    },
    {
      "name": "Michael Collins",
      "born_died_date": "born 20 Jan 1930"
    }
  ]
}
```

e. precipitation dataset (netCDF4 format)

```
latitude: float[50]
longitude: float[30]
time: float[100]
precipitation: float[50][30][100]
```

Figure 1: Three datasets that have different formats and layouts. Their data is truncated for readability. In the precipitation dataset, only the schema is shown.

2 MOTIVATING EXAMPLE AND PROBLEM REQUIREMENTS

In this section, we provide examples of diversely-structured data sources to illustrate the challenges of mapping them to RDF, the limitations of previous mapping methods, and the requirements of the generic mapping languages.

Figure 1a and 1b show a life expectancy table for the population of South Sudan[1] in matrix and row-based layouts, respectively. The original table (Figure 1a) contains observations (green matrix) of indicators (dark blue column) such as life expectancy. These observations were collected over several years (light blue row) for different gender (yellow row) and age groups (orange column). This original data can be represented in RDF using the Data Cube vocabulary[2] but manually generating this representation is cumbersome and it is difficult to generate consistently across multiple sources [1]. Specifically, this representation requires dealing with the complex, nested matrix layout, imputing missing values, such as the implicit years in the light blue row, and transforming values, such as converting age groups into a range of age values. Task-specific tools, such as XLWrap and M2, have been developed to map these types of datasets to RDF, however they cannot be used for all datasets.

Figure 1c and 1d show a second data source formatted in JSON that contains a list of artworks from the National Portrait Gallery (NPG). Each artwork also includes information about its creator and sitters. As XLWrap and M2 do not support JSON data, users have to rely on extensions of R2RML instead. These extensions are created to support mapping sources with various formats such as JSON, XML, RDBMS, etc. However, these extensions are limited to the layout conventions used in the Nested Relational Model (NRM). In the original dataset (Figure 1c), the records do not follow the conventions of the NRM because sitter names and sitter birth dates are stored in two separated arrays, whose values are associated by their orderings. To handle these scenarios, a different system has to be used to transform the structure of the source into the layout in Figure 1d (e.g., zip the two arrays to create one array) using transformation functions as in KR2RML [8].

Figure 1e shows an extreme example of a rainfall dataset, in which volume of rainfall (*precipitation*[x][y][z]) is recorded at a specific location (*latitude*[x], *longitude*[y]) and at a specific time (*time*[z]). This dataset is one of many scientific datasets that have customized layout for efficiently storing or manipulating the data. Since sharing scientific knowledge can accelerate new findings, it is important that we be able to easily map these datasets to RDF.

As no existing generic mapping language can handle all five examples above, users have to spend significant effort learning to use different tools for different kinds of sources. As we have discussed in the previous section, one reason is the existing generic languages lack the ability to express the layout of data sources. This poses a new requirement that the languages need to support. Together with the requirements from [2, 5, 6], we create a list of 6 important requirements that the generic mapping language should meet as follows.

R1 Map sources with heterogeneous formats and be extensible to new data formats.

R2 Map sources with heterogeneous layouts and be extensible to custom layouts.

R3 The language is uniform so that extending to new formats and layouts only requires changes in the implementation (e.g., adding plugins), but not in the language itself.

R4 Support interlinking across sources, e.g., generating links between individuals of different classes.

R5 Support cleaning and transforming data.

R6 Support general mapping RDF functionalities, e.g., specifying the data types and used ontologies, generating blank nodes.

Based on the above requirements, in the next section, we will describe D-REPR, a new mapping language, that addresses the above use cases and supports a broad set of capabilities.

[1] https://apps.who.int/gho/data/view.main.LT62270
[2] https://www.w3.org/TR/vocab-data-cube/

3 DATASET REPRESENTATION LANGUAGE

In this section, we describe our dataset representation language (D-REPR) and its usage in the context of a sample dataset in Figure 2. The sample dataset contains a CSV file that is similar to the life table in Figure 1a, except that the indicator column contains the indicator code. The dataset has another JSON file that has the indicators' definitions.

a. life table.csv

		2016	
Indicator	Age Group	Male	Female
LIFE_0035	<1 year	57.7	59.6
LIFE_0035	1-4 years	60.6	62.1

b. indicators.json

```
{
  "indicator": "LIFE_0035",
  "url": "http://apps.who/int/ ... /indicator.aspx?iid=35"
},
{
  "indicator": "LIFE_0029",
  "url": "http://apps.who/int/ ... /indicator.aspx?iid=29"
},
```

Figure 2: A sample life table dataset.

Modeling a dataset in D-REPR requires defining four basic components: resources, attributes of data sources, a semantic model that maps the attributes to properties of ontology classes, and rules to join values of these attributes.

In addition, many data sources have useful data invariants (e.g., missing values) or benefit from pre-processing functions that D-REPR also supports. Listing 1 shows an overview of the D-REPR syntax for modeling dataset. We use YAML as it is concise and human-friendly to write. However, the D-REPR specification can also be expressed in other languages such as JSON.

3.1 Resources and attributes of the dataset

Resources. The first step of defining a dataset is describing its resources. A dataset may consist of many, interdependent logical resources such as codebooks or data-definition dictionaries. These logical resources are defined under a section titled `resources`. Each resource is associated with a user-defined name and a format. In the sample dataset, we have two resources: a life table and a list of indicators, whose definitions are in Listing 2.

Different data formats have different ways of referring to data elements. For example, XML and JSON documents have XPath and JSONPath, respectively, or CSV and Spreadsheet can refer to cells by sheet name, row and column number. In addition, query results from relational and non-relational database such as graph databases can also be manipulated using well-known formats such as JSON or row-based table. Therefore, it is safe to view resource's data as a general tree structure similar to JSON tree. For example, a dataset in the NetCDF format can be viewed as a JSON object, where variables in the dataset are properties of the object, values of the variables (multi-dimensional arrays) are nested JSON arrays. Similarly, a CSV table can also be viewed as a nested JSON array.

Viewing resource's data as a tree has several benefits. First, it facilitates downstream tasks such as data cleaning to be independent of the data formats. Second, it enables users to use one single query path language such as JSONPath to refer to a resource's elements. Finally, it reduces the implementation effort as we only need to implement one path language per format. This is different from other generic mapping languages such as RML [2] or xR2RML [6] as they allow users to use various path languages via `rml:referenceFormulation`.

We use JSONPath expressions to select data regions in a resource. The query starts with `$` as the root of a resource. To select a child

```
resources:
    <resource_id>:
        type: <resource_type>
        # ..optional arguments depend on the resource type..
    # ..other resources..
[preprocessing]:
    - type: <function type>
      input:
          [resource]: <resource_id>
          path: <json_path>
      [output]: <new_resource_id>
      [code]: <code>
    # ..other transformation functions..
attributes:
    <attribute_id>:
        [resource_id]: <resource_id>
        path: <json_path>
        [unique]: false
        [missing_values]: [<value_0>, <value_1>, ...]
    # ..other attributes..
alignments:
    - type: <join_type>
      source: <attribute_id>
      target: <attribute_id>
      # ..optional arguments depend on the alignment type..
    # ..other alignments..
semantic_model:
    data_nodes:
        <attribute_id>: <class_id>--<predicate>[^^<data_type>]
        # ..other attributes..
    [relations]:
        - <source_class_id>--<predicate>--<target_class_id>
        # ..other relations..
    [subjects]:
        <class_id>: <attribute_id>
        # ..other subjects..
    [prefixes]:
        <prefix>: <iri>
        # ..other prefixes..
```

Listing 1: The D-REPR YAML syntax to describe a dataset. Optional properties are surrounded by brackets (e.g., `[missing_values]` is optional)

```
resources:
  life_tbl:
    type: csv
    delimiter: ","
  indicators:
    type: json
```

Listing 2: Resources' definition of the sample dataset

of the current element, we use `.<key>` or `[<key>]`. To select a range of child nodes, we use the array slice operator `[<start>:<end>:<step>]`. To select all child nodes, we use either `.*` or `[*]`. For example, indicator urls in the JSON resource can be retrieved by `$[*].url` or `$[:].url`, and observations (green cells) in the life table can be selected by `$[2:][2:]`.

Attributes. The next element in the D-REPR language is a block of `attributes` that designates specific data attributes found in the data source. Each attribute is first defined by providing a unique, human-readable name. Next, each attribute is associated with a spatial location where the attribute is physically located. The location includes both `resource_id` and `path`, which is a path expression to its values in the resource using the JSONPath.

In addition to specifying the location of an attribute, the D-REPR specification can optionally include particular values that should be interpreted as missing values (e.g., -999) in the `missing_values` property. It can also specify whether values of an attribute are unique by setting the `unique` property to be true (e.g., the *indicator* attribute contains all unique values). Listing 3 shows the definition of the attributes of the sample dataset.

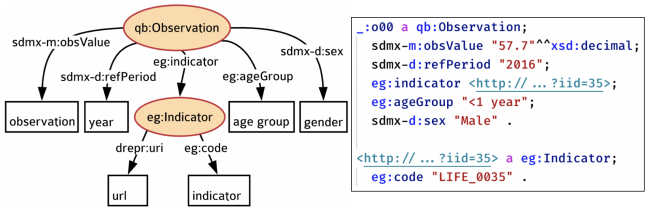
```
attributes:
  year:
    resource_id: life_tbl
    path: $[0][2:]
  gender:
    resource_id: life_tbl
    path: $[1][2:]
  indicator_column:
    resource_id: life_tbl
    path: $[2:][0]
  age_group:
    resource_id: life_tbl
    path: $[2:][1]
  observation:
    resource_id: life_tbl
    path: $[2:][2:]
  indicator:
    resource_id: indicators
    path: $.*.indicator
    unique: true
  url:
    resource_id: indicators
    path: $.*.url
    unique: true
```

Listing 3: Attributes' definition of the sample dataset

3.2 Semantic model of the dataset

After identifying resources and source attributes, we specify the semantic model for mapping data to RDF. The semantic model is a graph, in which internal nodes are ontology classes, leaf nodes are attributes (also called data nodes) and edges are ontology predicates. The model can be defined under the `semantic_model` block in Listing 1. Figure 3 depicts a semantic model and a subset of the generated RDF triples of the sample dataset.

The first step in defining the semantic model is to associate each attribute with a semantic type, which is a pair of an ontology class C and an ontology predicate p, denoted as $\langle C, p \rangle$. This creates predicate-object pairs for each instance of the ontology class C. The subject of each instance is created from the other attribute whose semantic type is of the same class C and uses a particular predicate `drepr:uri`. If there is no such attribute, then the instances will be blank nodes. We sometimes refer to attributes that are mapped to properties of a class as attributes of the class for short. For instance in Figure 3, the semantic type of the attribute *observation* is defined as `"qb:Observation:1--sdmx-m:obsValue^^xsd:decimal"`




```
semantic_model:
  data_nodes:
    observation: qb:Observation:1--sdmx-m:obsValue^^xsd:decimal
    year: qb:Observation:1--sdmx-d:refPeriod
    age_group: qb:Observation:1--eg:ageGroup
    gender: qb:Observation:1--sdmx-d:sex
    indicator: eg:Indicator:1--eg:code
    url: eg:Indicator:1--drepr:uri
  relations:
    - qb:Observation:1--eg:indicator--eg:Indicator:1
  prefixes:
    qb: http://purl.org/linked-data/cube#
    sdmx-m: http://purl.org/linked-data/sdmx/2009/measure#
    sdmx-d: http://purl.org/linked-data/sdmx/2009/dimension#
    eg: http://example.org
  subjects:
    "qb:Observation:1": observation
    "eg:Indicator:1": indicator
```

Figure 3: A semantic model of the sample dataset

means that it is associated with a pair of ontology class and predicate ⟨`qb:Observation`, `sdmx-m:obsValue`⟩ and with the data type `xsd:decimal`. Notice in the definition, we have the ontology classes appended with an extra number (`qb:Observation:1`), which are used as identifiers to uniquely identify the class node. The reason is that there can be multiple class nodes that have the same label in the semantic model. For instance, in the NPG museum dataset, we have two `foaf:Person` classes, one is for creators of the artworks, one is for sitters in the artworks.

In addition to semantic types, semantic relations between instances are determined by edges (labeled with ontology predicates) between their classes. We can specify the semantic relations under the `relations` property. For example, the relationship `eg:indicator` of the `qb:Observation` and `eg:Indicator` is specified as `"qb:Observation:1--eg:indicator--eg:Indicator:1"`. Together with the semantic types defined in the `data_nodes` section, they form a graph of the semantic model as in Figure 3.

Moreover, as the IRIs of ontology classes and predicates are quite wordy, D-REPR allows users to use prefixed names similar to the Turtle language. The list of prefix labels and their corresponding IRIs are defined under the `prefixes` property.

Finally, D-REPR and other mapping languages assume that for each ontology class, there is at least one attribute whose values are in one-to-one correspondence with the instances of the class. In other words, we can iterate through the values and generate instance one by one. We say this attribute is the subject attribute of the class. In D-REPR, these subject attributes can usually be inferred automatically as shown in Section 4.1. This eases the task of updating the semantic model of the dataset, which occurs quite frequently in publishing datasets as 5-star Linked Data [3]. However, users can explicitly specify them in the `subjects` property (Figure 3).

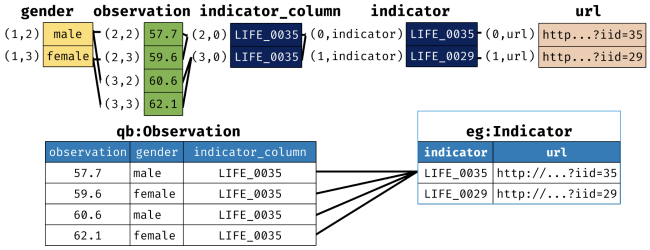
3.3 Joining the values of attributes of a dataset

After users define the attributes in the dataset, the values of each attribute are retrieved and represented as an array. Because there is no alignment between these arrays, to generate instances of an ontology class, we need to join together values of the attributes that were mapped to the properties of a class. Figure 4 illustrates how attributes in the sample dataset should be joined. In particular, *observation* and *gender* are joined by their column position, *observation* and *indicator_column* are joined by their row position, and *indicator_column* and *indicator* are joined by their values.

In D-REPR, a list of joins between the attributes is declared under the section titled `alignments` (Listing 1). Each join is required to define three properties, which are join type (`type`) and the two attributes involved in the function (`source` and `target`). D-REPR currently supports two join functions: joining by values (called value join) and by value position in the resources (called position join). The types for the two functions are `value` and `dimension`, respectively.

Value join is a traditional equity join in SQL. A value join only requires a source and target. Figure 4 shows an example of a value join between the *indicator_column* and *indicator* attributes.

Position join combines values if their positions are matched. A position of an attribute's value is a path to the value in the resource, and we refer to each item at index i in the path as dimension i. For example, the age group "1-4 years" is at position p = `[3, 1]`, the value




```
alignments:
  - type: dimension
    source: observation
    target: gender
    aligned_dims: [{ source: 1, target: 1 }]
  - type: dimension
    source: observation
    target: indicator_column
    aligned_dims: [{ source: 0, target: 0 }]
  - type: value
    source: indicator_column
    target: indicator
  - type: dimension
    source: indicator
    target: url
    aligned_dims: [{ source: 0, target: 0 }]
  # .. more joins
```

Figure 4: Joining between the attributes in the sample dataset. Similar joins between ⟨*observation*, *year*⟩ and ⟨*observation*, *age group*⟩ were omitted for readability

of dimension 0 is $p[0] = 3$ and the value of dimension 1 is $p[1] = 1$. Given a set of pairs of aligned dimensions $S = \{(x_1, y_1), (x_2, y_2), ...\}$, in which x_i, y_i are dimensions of attribute x and y, respectively, two positions p_x and p_y of x and y are considered as matched when $\forall (x_i, y_i) \in S : \frac{p_y[y_i] - y_i^{\text{start}}}{y_i^{\text{step}}} = \frac{p_x[x_i] - x_i^{\text{start}}}{x_i^{\text{step}}}$, where x_i^{start} and x_i^{step} are start and step of a range index of dimension x_i. A position join takes an extra property, which is the set S in `aligned_dims` property. An example of a position join is a join between *indicator* and *url* in Figure 4, where the aligned dimensions are `[{ source: 0, target: 0}]`, which means a value of *indicator* at position `[1, indicator]` only matches with a value of *url* at `[1, url]` not at `[0, url]`.

Given a list of joins, instances of an ontology class are created by first creating a single column table of its subject attribute. Then, we sequentially join every other attribute of the class to the table using the join function between the attribute and the subject attribute. Recall that values of the subject attribute are assumed to have a one-to-one correspondence to instances of the class. Therefore, the size of the joined table does not change after each attribute is added and each row in the table contains one instance of the class. For example, the table of the class `qb:Observation` is created by joining the subject attribute *observation* and *gender*, then joining *indicator_column* to the table (Figure 4). This requires users to define $n-1$ joins between

subject attributes and other attributes for n attributes. However, as we discussed in the previous section, subject attributes are usually inferred automatically. Therefore, users may inadvertently define join functions of non-subject attributes. For example, in the sample dataset, a user might define join functions between ⟨*observation*, *gender*⟩ and ⟨*gender*, *year*⟩. A join function between ⟨*observation*, *year*⟩ is missing, but D-REPR can complete missing join functions between subject and non-subject attributes, which is described in Section 4.1. As a result, users can focus on specifying the alignments between $n-1$ pairs of attributes that explain the layout of the dataset. If D-REPR cannot infer missing functions, the system will ask users to provide the correct definitions.

3.4 Data cleaning and data transformation

We use transformation functions to facilitate data cleaning and data transformation. D-REPR defines a list of transformation functions under the `preprocessing` keyword as shown in Listing 1. These functions are executed sequentially before the mapping process starts. Each function is applied to a data region specified in the `input` property using the JSONPath. If the `output` property is also defined, the function runs and produces new data stored in the new resource. Otherwise, the function runs and mutates data in the `input` region. The optional `code` property allows users to write one-time-use ad-hoc data cleaning code. Transformation functions in external libraries can also be used by importing them to the D-REPR processor and specifying their IDs in the `type` property. Hence, users can reuse transformation functions across datasets.

One concrete example of the above transformation functions is the mapping function implemented in the prototype of the D-REPR processor. The function maps each data element in the `input` property to a new data element using python code defined in the `code` property. The python code takes three arguments ⟨`value`, `index`, `context`⟩ and computes and returns a new value. The `value` variable is the value of the current data element, the `index` variable is the element's position, and the `context` variable is a python helper object that has some methods to query resource data. Listing 4 shows an illustration of the mapping function for the sample dataset. In the example, we use a method `context.get_left_value(index)` to access to the element on the left of the current element (e.g., given the `index` is `[0, 3]`, the method returns value at position `[0, 2]`).

```
preprocessing:
  - type: pmap
    input:
      resource: life_tbl
      path: $[0][2:]
    code: |
      if value == "":
        return context.get_left_value(index)
      return value
```

Listing 4: A transformation function for the sample dataset, it replaces empty light blue cells (years) with values in the left cells

4 IMPLEMENTATION AND EVALUATION

In this section, we describe one possible implementation of the D-REPR specification and evaluate the ability of D-REPR to map real-world datasets in different formats and layouts. A prototype of the D-REPR processor[3] is available online for evaluation. We compare the runtime between the prototype of the D-REPR processor and popular processors of R2RML extensions to evaluate the ability of the D-REPR processor to handle large datasets.

4.1 Algorithm for RDF generation

In this implementation, the D-REPR processor generates RDF in three steps. First, the system create execution plans that determine the order for generating instances of the ontology classes in the semantic model. In case users do not provide the subject attributes, the system will infer the subject attributes of these classes and determine missing join functions between the subject attributes and the non-subject attributes. The next step is to execute preprocessing functions one by one. Finally, the system iterates through each ontology class in the semantic model and outputs their instances.

To generate instances of a class, we need to join the values of the attributes of the class. A join function f between attribute a_i and a_j ($f : a_i \rightarrow a_j$) has an abstract interface as follows:

$$f(d_i \in V_i) = \{d_j | d_j \in V_j\}$$

in which V_* is a list of pairs of values and value positions of attribute a_*. The function f takes an element d_i of V_i of attribute a_i and return a set of elements of a_j matched with d_i. For example, in the sample dataset, V_{gender} of attribute *gender* is `[("male", [1, 2]), ("female", [1, 3])]`. Given a value d_i = `("male", [1, 2])` $\in V_{\text{gender}}$, the join function f : *gender* $\rightarrow$ *observation* will return a set `{(57.7, [2, 2]), (60.6, [3, 2])}`. This abstract interface enables us to hide the details of the join function (such as value join or position join) and its implementation (such as hash join or sort-merge join), and enables us to incorporate new join functions easily.

Given a subject attribute and a list of join functions between the subject attribute and other attributes of a class, we can generate instances of the class using Algorithm 1. Specifically, the algorithm iterates each value of the subject attribute to create one instance and uses the join functions to get properties of the instance (lines 4 - 6).

Inferring subject attributes. The next step is to infer a subject attribute of a class and missing join functions between attributes. Let $\mathcal{A} = \{a_1, a_2, ..., a_n\}$ be a set of attributes of class C. We have $a \in \mathcal{A}$ is a subject attribute of C, if and only if for every pair of attribute a and $a_i \in \mathcal{A}$, the degree of relationship is one-to-one or many-to-one (i.e., the join function f between a and a_i satisfies: $\forall d_i \in V : |f(d_i)| \leq 1$). Indeed, if we join all $a_i \in \mathcal{A}$ with a, every row in the result table is unique, and the table contains all possible instances of the class C in the dataset. Therefore, a is the subject attribute of C.

If there is no such attribute in $\mathcal{A}$, we may try to pick another attribute a' in the dataset such that its degree of relationship with every attribute $a_i \in \mathcal{A}$ is one-to-one or many-to-one. This condition is similar to the above rule. However, with a', the joined table

[3] https://github.com/usc-isi-i2/d-repr

Algorithm 1: Generating Instances Of a Class

Input: A subject attribute of class C: a
A list of values and their positions of a: V_a
A hash map from an attribute a_i of C to a join function between a and a_i: $\mathcal{F} = \{a_i \rightarrow f\}$
Output: List of instances of class C

```
instances ← []
for d ← V_a do
    instance ← {}
    for f ← F do
        a_i ← target variable of f
        instance.a_i = f(d)
    instances.append(instance)
return instances
```

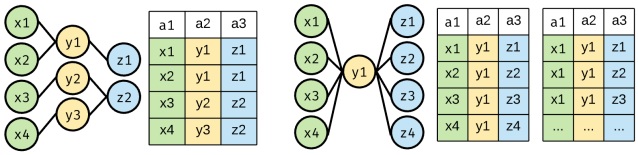
now has an extra column a'. To obtain the table of instances of C, we have to remove column a', and potentially filter out duplicated instances in the table. If there are no duplicates, a' can be chosen as the subject attribute of C.

We can determine the cardinality between two attributes based on the join function between them, conservatively. If attributes a_1 and a_2 are joined by value, and values of a_1 or a_2 are unique, then their degree of relationship is either one-to-* or *-to-one. If none of them are unique, their cardinality is many-to-many. For example, in the sample dataset, the cardinality between *indicator_column* and *indicator* is many-to-one. The cardinality of position join is computed based on non-aligned dimensions between two attributes. For example, in the sample dataset, *observation* (location: `$[2:][2:]`) and *year* (location: `$[0][2:]`) have one aligned dimension (dimension 1). Because the dimension 0 of *observation* is specified by the array slice operator (`[2:]`), their cardinality is many-to-one. However, the degree of relationship between *year* and *gender* (location: `$[1][2:]`) is one-to-one as their first dimensions are specified by array index operators (`[0]` and `[1]`).

Chaining join functions between attributes. Suppose that users define join functions of $\langle a_1, a_2\rangle$ and $\langle a_2, a_3\rangle$. However, they do not specify any rule to join attributes a_1 with a_3. If the cardinality of $\langle a_1, a_2\rangle$ is either one-to-many or one-to-one, or the cardinality of $\langle a_2, a_3\rangle$ is either many-to-one or one-to-one, we can compute the join function of a_1 and a_3 as a chained join function that combines of a_1 and a_2 to obtain table $T_1(a_1, a_2)$, before joining a_3 with T_1. The intuition is that it is the only possible alignment between values of a_1 and a_3 which is consistent with the two defined join functions. The chained join is illustrated in Figure 5a. An example in Figure 5b shows that we cannot infer the join function of a_1 and a_3 when the cardinality of $\langle a_1, a_2\rangle$ is many-to-one and $\langle a_2, a_3\rangle$ is one-to-many.

4.2 Evaluation

To assess the capabilities of D-REPR to model datasets in different formats and layouts, we crawled 10,000 public datasets on data.gov, then filtered and randomly sampled 700 datasets in CSV, JSON, XML, Spreadsheet and NetCDF formats. Within these 700 datasets, we manually selected distinct datasets of different formats or layouts,



(a) Only one possible table created from the join between attributes a_1 and a_3

(b) Cannot infer how to join a_1 and a_3 because there is more than one possible result

Figure 5: Chaining join functions between attributes $\langle a_1, a_2\rangle$ and $\langle a_2, a_3\rangle$

as datasets in the same format and layout will have similar D-REPR models. The resulting datasets are diverse[4]. For example, some of them have complex layouts, such as nested JSON arrays with column definitions in another property (Figure 6a). Data in some datasets also need cleaning (e.g., date-time reformatting or extracting hierarchical structure from a column (Figure 6b)). In order to handle all of the datasets, a mapping language is required to satisfy the 6 requirements in Section 2. We are able to build D-REPR models for all of them, demonstrating that our language meets the requirements.

To verify that the D-REPR processor can handle large datasets, we compare the runtime of our prototype with two popular processors of R2RML extensions (KR2RML [8] and Morph[5]) on the task of mapping large CSV files (NRM layout). All CSV files contain people information (e.g., name, phone and address). The only difference between these files is the number of records. Our experiments are run on a Macbook Pro, Intel i7-7660U with 16GB RAM. The average runtime is reported in Table 1. The runtime of the D-REPR processor increases linearly with the number of records. On average, it can generate 1.3 million triples per second, which is 15 times faster than KR2RML. The reasons that D-REPR is fast are its core engine is implemented using the Rust language; RDF triples are

[4]The experiments are available at http://purl.org/tty1/drepr-exp
[5]https://github.com/oeg-upm/morph-rdb

a. Children and Family Health

```
{
  "columns": [
    {"name": "teenbir10", "description": "Teen Birth Rate ... (2010)"},
    {"name": "teenbir11", "description": "Teen Birth Rate ... (2011)"},
    ...],
  "data": [
    [..., "Allendale/Irvington/S. Hilton",   "55.0", "58.1", ...],
    [..., "Beechfield/Ten Hills/West Hills", "42.8", "21.4", ...],
    ...],
  ...
}
```

b. Sugar production by sugar beet and sugarcane processors

FY 2008	JAN	FEB	MAR	APR	MAY	JUN
From domestic sugar beets	661,586	485,126	423,775	337,473	216,526	82,987
From imported sugar beets	0	37,160	0	0	0	0
Subtotal	661,586	522,287	423,775	337,473	216,526	82,987
Cane production:						
Florida	321,414	253,438	242,560	92,302	47,237	0
...	...					
Subtotal	378,919	283,190	289,237	108,826	68,504	30,903
Total	1,040,505	805,476	713,012	446,298	285,030	113,889

Figure 6: Examples of datasets with complex layouts in data.gov. In figure (a), data is in nested JSON array where each column definition is defined in the *columns* property (40 columns). Figure (b) is a dataset that has hierarchical structure (green column).

Table 1: Average running time (ms) of D-REPR and popular processors of R2RML extensions

	Number of records				
Tools	5,000	10,000	20,000	40,000	80,000
D-REPR	33.44	69.84	132.00	267.50	551.24
KR2RML	1368.00	1776.33	3276.66	4990.33	8305.33
Morph	4812.00	14949.66	65961.33	-	-

generated using a streaming approach; and it leverages properties of attributes to generate an efficient execution plan (e.g., missing value checks are not needed if there are no missing values).

5 RELATED WORK

There are many proposed mapping languages or tools to transform data sources into RDF. W3C communities have listed many of them on their website. We can classify them into three different groups.

The first group of tools is specific to certain formats and structures such as bibtex2rdf, or email2rdf. As they are designed for one particular type of source, they may be the most effective tools to generate RDF for that source, but they do not apply to other source formats.

The second group is languages that work with tabular data. It is hard to convert tabular data as they have arbitrary ways of storing and representing data. XLWrap [4] is the language designed for mapping spreadsheets with various layouts to RDF. Users first describe a template graph similar to a semantic model, in which leaf nodes (known as data nodes) are bound to cells in the table. Then, they define a transformation operation that shifts rows and columns in the spreadsheet such that after every shift, the data nodes in the template graph are bound to new values and produces a new instance. Corcho et al. [7] present the M^2 language, which uses the Manchester Syntax for converting data from spreadsheets into OWL. The language is less verbose than XLWrap and also able to handle many different tabular layouts. The main drawback of these mapping languages is although they can map datasets with heterogeneous layouts, they are limited to only tabular formats such as CSV or spreadsheets.

The third group contains generic mapping languages for integrating heterogeneous data sources. Dimou et al. [2] propose RML, an extension over R2RML, for mapping data sources with different formats and interlinking between sources. In particular, users describe logical sources and their formats, iterators to iterate through records in the sources, and how to map each record's attributes. Michel et al. [6] introduce xR2RML that extends R2RML and RML to support different types of databases from relational databases to non-relational databases such as Cassandra or MongoDB. Slepika et al. [8] present another extension of R2RML, KR2RML. Similar to xR2RML, it supports heterogeneous hierarchical sources. KR2RML also allows users to perform data cleaning or modifying data structures so that users can model noisy hierarchical data sources such as the NPG museum dataset (Figure 1c). Lefrançois et al. [5] develop SPARQL-Generate based on a SPARQL that allows querying a combination of RDF and non-RDF sources. Although the listed languages are capable of handling data sources with heterogeneous formats, the main difference between them and D-REPR is that they only work with the NRM layout.

6 CONCLUSION AND FUTURE WORK

In this paper, we presented a novel dataset representation language, D-REPR, for mapping data to RDF. To handle datasets with heterogeneous formats, the language enables users to use JSONPath to select values of each dataset attribute. With datasets in different layouts, it allows users to describe join rules to combine the values, which capture the data alignments within each layout. Users can specify ontology classes and predicates that the data should be mapped to using semantic models. The language also supports data cleaning and data transformation using pre-processing functions. The language is designed to be extensible. We can add a new data format by implementing the JSONPath for the format. We can also incorporate a new join rule by implementing the abstract interface of the join function.

In future work, we plan to improve the efficiency of D-REPR's engine for transforming data to RDF by partitioning resource data into small batches and executing the mapping operation in parallel. We also plan to develop machine learning approaches for creating and suggesting D-REPR models. Finally, we plan to build a web interface that can be integrated into open data platforms in which users can register a dataset and build a representation for it. That will encourage people to publish many more data sources in RDF.

ACKNOWLEDGMENTS

This research was sponsored by the Army Research Office (ARO) and Defense Advance Research Projects Agency (DARPA) under award W911NF-18-1-0027. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARO, DARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. We thank the anonymous reviewers for their helpful comments on the paper.

REFERENCES

[1] Karin Becker, Shiva Jahangiri, and Craig A. Knoblock. 2015. A Quantitative Survey on the Use of the Cube. In *Proceedings of the 3rd International Workshop on Semantic Statistics*.

[2] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. 2014. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. *7th Workshop on Linked Data on the Web, Proceedings* 1184 (2014).

[3] Craig A. Knoblock, Pedro Szekely, Eleanor Fink, David Newbury Duane Degler, Robert Sanderson, Kate Blanch, Sara Snyder, Nilay Chheda, Nimesh Jain, Ravi Raju Krishna, Nikhila Begur Sreekanth, and Yixiang Yao. 2017. Lessons Learned in Building Linked Data for the American Art Collaborative. In *ISWC 2017 - 16th International Semantic Web Conference*.

[4] Andreas Langegger and Wolfram Wöß. 2009. XLWrap — Querying and Integrating Arbitrary Spreadsheets with SPARQL. In *ISWC '09 Proceedings of the 8th International Semantic Web Conference*. 359–374.

[5] Maxime Lefrançois, Antoine Zimmermann, and Noorani Bakerally. 2017. A SPARQL extension for generating RDF from heterogeneous formats. In *European Semantic Web Conference*, Vol. 10249. 35–50.

[6] Franck Michel, Loïc Djimenou, Catherine Faron Zucker, and Johan Montagnat. 2015. Translation of relational and non-relational databases into RDF with xR2RML. In *11th International Confenrence on Web Information Systems and Technologies (WEBIST'15)*. 443–454.

[7] Martin J O'Connor, Christian Halaschek-Wiener, and Mark A Musen. 2010. M2: A Language for Mapping Spreadsheets to OWL.. In *OWLED*, Vol. 614.

[8] Jason Slepicka, Chengye Yin, Pedro Szekely, and Craig A. Knoblock. 2015. KR2RML: An Alternative Interpretation of R2RML for Heterogenous Sources. In *Proceedings of the 6th International Workshop on Consuming Linked Data (COLD 2015)*.