Linking application and semantic data with RDF Lens

Arthur Vercruysse[1,*], Julián Rojas Meléndez[1,*] and Pieter Colpaert[1]

[1]IDLab, Department of Electronics and Information Systems, Ghent University – imec

Abstract

Linked Data is commonly regarded as an unfriendly data structure to be directly used by application developers. The (often unknown) triple-based structure of RDF graphs causes developers to struggle to extract the triples of interest and translate them into the object-like structure needed for their application. A generic, composable and reusable way to look into RDF graphs as plain objects would remove an important barrier for integrating Linked Data in all facets of application logic. We propose RDF Lens, a library based on ideas from the Haskell lens library that allows for composable and reusable data extraction units, called lenses. Value is shown by implementing a lens that generates a new lens based on a SHACL shape that extracts the semantic data into the desired plain object. Abstracting data extraction at the lens level allows for mixed extraction: using both custom extraction and declarative extraction, which could increase ease of use and reusability. The current implementation is a proof of concept that defines how to extract data from an RDF graph in JavaScript applications but does not allow (yet) writing or altering linked data with the same lenses. Future work would allow for creating and updating Linked Data in the same elegant way.

Keywords

RDF, SHACL, Functional programming, Lenses, JavaScript, Haskell

1. Introduction

When processing RDF data, a developer needs to deal with a list of interlinked triples that conform to a graph. The promise is that the added complexity of dealing with a graph can lead to higher interoperability as the IRIs used in named nodes in any position of the triple can be shared across datasets and systems. The contract between data and application code is loosened, as developers should not code against specific objects, but against graph patterns, allowing more systems to be connected.

Coding against such graph patterns is more complex than coding against structured data (e.g, JSON objects) and may require following non-traditional programming paradigms to build efficient and reusable code [1]. While applications processing JSON-like objects already get the data in the right structure, applications processing RDF data need to write additional logic to create the structures they need to process data further. This

SEMANTiCS'24: International Conference on Semantic Systems, September 17–19, 2024, Amsterdam, NL

*Corresponding author.

arthur.vercruysse@ugent.be (A. Vercruysse); JulianAndres.RojasMelendez@UGent.be (J. R. Meléndez); pieter.colpaert@ugent.be (P. Colpaert)

https://pietercolpaert.be (P. Colpaert)

0000-0003-3467-9755 (A. Vercruysse); 0000-0002-6645-1264 (J. R. Meléndez); 0000-0001-6917-2167 (P. Colpaert)

becomes time-consuming and may lead to code duplication across the codebase, as similar logic might be needed for slightly different cases.

In the JavaScript ecosystem, libraries like Linked Data Objects [2], LDFlex [3], RDF Object[1], or SimpleRDF[2] rely on JSON-LD contexts, which does not allow for type-checking on all object properties. Another type of library, ts-rdf-mapper[3] builds code that interacts with an RDF dataset based on annotations on the type definitions in the program itself. This approach is tightly coupled with the code base making it very difficult to reuse in different applications, compared to reusing the same JSON-LD context in multiple programs.

We aim to build a framework that allows to *compose* different pieces of data processing logic together, potentially *reused* from earlier tasks, to deliver the structures needed for an application. In that respect, the functional programming concept of data *Lenses* [4], as implemented in the Haskell lens library[4], defines precisely such composable and reusable data structures. In Haskell, lenses are used to view, update or create (deeply) nested records. Without lenses, developers need to destructure entire objects and recompose them after each operation, similarly as it is needed when dealing with complex RDF graphs.

This paper shows how the idea of Haskell lenses can be applied to RDF. With lenses, extracting complex objects from a graph becomes easier due to the composability and typing support that lenses can provide. As the main use case, we show how to derive a new lens based on a SHACL shape, that in turn can extract objects according to the shape definition.

In the remainder of this paper, the second section provides a brief description of the Haskell lens concept. The third section shows how and why we apply the idea of lenses on RDF graphs. The next section describes the use case where we can derive a lens from a SHACL shape. The last section presents the conclusion and prospective future work.

2. Haskell lenses

In Haskell, working with structured data entails that for each modification on a nested value, a developer needs to destructure the object and reconstruct it back after performing the modification. This leads to unreadable and unmaintainable code. A Haskell lens aims to facilitate such procedures by lifting an operation over a value, to an operation over an object, also known as monads applied over data [5].

A lens is built by combining a getter and a setter function over a defined type. With a lens, is possible to perform operations over a data structure such as [6]:

- Access a subpart (view)
- Alter the whole by modifying a subpart (update)
- Merge the lens with another lens to get a deeper view (composition)

[1] https://github.com/rubensworks/rdf-object.js
[2] https://github.com/simplerdf/simplerdf
[3] https://github.com/artonio/ts-rdf-mapper
[4] https://hackage.haskell.org/package/lens

For example, take a data structure representing a `Line` type composed by a start and an end `Point` type. Increasing the x coordinate of the starting `Point` by one requires destructuring and reconstructing the entire `Line` object, as shown in Listing 1. In this example, a `Point` only has two coordinates, however, the code complexity increases quickly with more complex subtypes.

Lenses abstract how to access a nested object away and make it composable. Listing 1 also shows the same function implemented differently using lenses. The `over` function applies the `(+1)` after accessing the field and returns a modified object. That function is predefined in the Lens library, `start` and `x` are lenses generated by the library.

```haskell
data Point = Point { _x :: Double,    _y :: Double  }
data Line  = Line  { _start :: Point, _end :: Point }

-- Deconstruct to access the values, return a reconstructed object
shiftStartByOne (Line (Point x y) end) = Line (Point (x + 1) y) end
-- `over` applies a function over a value after accessing the value using the lens
shiftStartByOne' = over (start . x) (+ 1)
```

Listing 1: Type declaration of a `Line` and a function that increases the x coordinate by one, implemented naively and with a Lens.

3. RDF Lens

RDF Lens[5] borrows the idea of a Haskell lens. A developer can look at the same graph with different lenses to see different data. This implementation only covers the `get` functionality: extracting the data and creating a new derived object.

Often the starting type of an RDF lens is (G, x) where $G = (V, E) \wedge E \subseteq \{\{s, p, o\} | s, p, o \in V\} \wedge x \in V$ (called `Cont` in code). RDF Lens includes some basic lenses that can be combined into more complex ones. The *predicate lens* (`pred`), for example, converts `Cont` into `Array<Cont>`, matching all objects after following that predicate, thus taking a step in the graph. Combining two of these predicate lenses and combining the result, creates a lens that can extract a typed object `Point` with an X and a Y coordinate (see Listing 2). The lenses are composable, Listing 2 shows that a lens that extracts a `Line` can be composed from Point lenses.

```javascript
const EX         = "http://ex.org";
const xField     = pred(`${EX}/field_x`).map(({id}) => ({x: id.value}));
const yField     = pred(`${EX}/field_y`).map(({id}) => ({y: id.value}));
const pointLens  = xField.and(yField).map(point => Object.assign(...point));

const startField = pred(`${EX}/start`).one().then(pointLens).map(start => ({start}));
const endField   = pred(`${EX}/end`).one().then(pointLens).map(end => ({end}));
const lineLens   = startField.and(endField).map(points => Object.assign(...points));
```

Listing 2: A lens that extracts a `Point`, with an X and a Y coordinate from an RDF graph and a Lens that extracts a line from two points.

[5]The implementation is available under the MIT license on github (https://github.com/ajuvercr/rdf-lens).

4. SHACL-based object extraction

To demonstrate the utility of RDF Lens, we implemented a lens that creates a lens from a SHACL shape [6]. We extract each defined property and interpret that property according to the `sh:datatype` or `sh:class`. If a `sh:class` is defined it will recursively apply a lens that corresponds to that class. The resulting lens results in a dictionary, each field name uses the value of `sh:name`. SHACL properties that define cardinality are also leveraged. `sh:minCount` determines if a field is required or not, and `sh:maxCount` dictates whether a single value or an array is to be expected. An example defining a Line can be seen at Listing 3.

With RDF Lens is possible to combine the SHACL-derived lenses with custom lenses. Listing 4 shows how to link a custom lens, with the undefined `CBD` class. In this case, the lens extracts the Concise Bounded Description (CBD)[7] set of triples, starting from the current term. This lens is provided by the RDF Lens library but is possible to combine externally defined lenses. Another predefined lens defined by RDF Lens, extracts a given SHACL Path and results in a lens itself.

```
[] a sh:NodeShape;
    sh:targetClass <Point>;
    sh:property
        [ sh:name 'x'; sh:path <x>; sh:datatype xsd:float; sh:maxCount 1 ],
        [ sh:name 'y'; sh:path <y>; sh:datatype xsd:float; sh:maxCount 1 ].

[] a sh:NodeShape;
    sh:targetClass <Line>;
    sh:property
        [ sh:name 'start'; sh:path <start>; sh:class <Point>; sh:maxCount 1 ],
        [ sh:name 'end';   sh:path <end>;   sh:class <Point>; sh:maxCount 1 ],
        [ sh:name 'quads'; sh:path ();      sh:class <CBD>;   sh:maxCount 1 ].
```

Listing 3: A SHACL shape that defines a shape for a `Line` and a `Point`. This shape can be interpreted by RDF Lens to extract a `Line` from RDF data.

5. Conclusion and future work

In this paper, we introduced RDF Lens as a developer tool to transform RDF triples into a structure that can be used by an application. The tool is available as an open-source library.

The idea that Haskell and RDF have a similar issue, resulted in the creation RDF lens (for the JavaScript ecosystem). So far, we limited the scope for this proof of concept to only extract objects from an RDF graph, and we do not allow yet for altering or generating RDF from objects. This already resulted in a promising tool, and these limitations will be addressed in future work.

[6]The code is available in the repository: https://github.com/ajuvercr/rdf-lens/blob/master/src/shacl.ts
[7]https://www.w3.org/submissions/CBD

```javascript
import { NamedNode, Parser } from "n3";
import { CBDLens, extractShapes } from "rdf-lens";
const turtle = `
  <MyPoint> <start> [ <x> 5; <y> 6; ];
            <end>   [ <x> 0; <y> 7].
`;

const quads = new Parser().parse(turtle);
const shapes = extractShapes(shapeQuads);
shapes.lenses["CBD"] = CBDLens; // Externally combined lens

const lens = shapes.lenses["Line"]; // The lens that extracts a line
const point = lens.execute({id: new NamedNode("MyPoint"), quads});

// { "start": {"x": 5, "y": 6}, "end": {"x": 0, "y": 7}, "quads": [/* <6 quads> */] };
console.log(point);
```

Listing 4: Javascript code example combining SHACL-derived and custom lenses.

As RDF Lens is already used today in our tooling to facilitate dealing directly with RDF data in JavaScript applications (both client- and server-side), we aim to support a broader adoption of it in the RDF developer community. We hope this will be the start of an ecosystem of reusable and composable RDF lenses.

References

[1] S. Staab, S. Scheglmann, M. Leinberger, T. Gottron, Programming the semantic web, in: The Semantic Web: Trends and Challenges, Springer International Publishing, Cham, 2014, pp. 1–5.

[2] J. Morgan, Linked data objects (ldo): A typescript-enabled rdf devtool, in: The Semantic Web – ISWC 2023: 22nd International Semantic Web Conference, Athens, Greece, November 6–10, 2023, Proceedings, Part II, Springer-Verlag, Berlin, Heidelberg, 2023, p. 230–246. doi:10.1007/978-3-031-47243-5_13.

[3] R. Verborgh, R. Taelman, LDflex: a read/write Linked Data abstraction for front-end Web developers, in: Proceedings of the 19th International Semantic Web Conference, volume 12507, 2020, pp. 193–211. doi:10.1007/978-3-030-62466-8_13.

[4] R. Lemmer, Haskell Design Patterns, Packt Publishing Ltd, 2015.

[5] P. Wadler, Comprehending monads, in: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP '90, 1990, p. 61–78. doi:10.1145/91556.91592.

[6] S. L. Nita, M. Mihailescu, Lens, Apress, Berkeley, CA, 2019, pp. 145–151. doi:10.1007/978-1-4842-4507-1_20.