Scalable and Reactive Programming for Semantic Web Developers

Jean-Paul Calbimonte

LSIR Distributed Information Systems Lab, EPFL, Switzerland.
`firstname.lastname@epfl.ch`

Abstract. The Semantic Web brings a powerful set of concepts, standards and ideas that are already changing the shape of the Web. However, in order to put these notions into practice we need to translate them into code. That is why the broad notion of *programming the Semantic Web* is important, even if it remains challenging to provide the appropriate tools for developers to implement these ideas. In this work, we present our experience in a series of common Semantic Web programming tasks, and how we improve the developer experience and productivity by providing a simple Domain Specific Language (DSL) in Scala. We exemplify its use through fundamental tasks such as dealing with RDF data and ontologies, or performing reasoning and querying using JVM languages. In particular, we show how we can profit from the intrinsic extensibility of the Scala language to simplify these tasks using a custom DSL, at the same time that we introduce principles of reactivity and scalability in the code.

1 Introduction

The Web is progressively incorporating elements, ideas and standards that emerged in the Semantic Web community in the latest years. Developers are at the forefront of these initiatives, creating semantic vocabularies, defining ontologies, manipulating RDF annotations, or building complex knowledge graphs [4]. While there is an increasing number of tools and libraries to help programmers to accomplish these tasks, we believe that there is large room for improvement, specially in the APIs of common Semantic Web libraries. In this paper we present our experience with some well-known JVM-based libraries for managing RDF, OWL and SPARQL, and the pitfalls we find when using their APIs. Moreover, we propose using a simple Domain Specific Language (DSL) using the extensibility features of the Scala language. We have chosen Scala as it is *scalable* in the sense that it is designed to "grow with the demands of its users" [3]. More concretely, it allows customizing and adapting APIs, through the usage of macros, implicits and other constructs, which greatly simplify the development process and remove unnecessary boilerplate code. Furthermore, we show how we can introduce constructs such as *Futures* and *Actors*, which provide a framework for building reactive and scalable applications.

The paper is organized as follows: In Section 2 we identify common issues in RDF manipulation and how our DSL can help alleviating them. In Section 3 we exemplify the use of Futures for asynchronous evaluation in SPARQL queries. In Section 4 we showcase the use of Actors for reactive programming on RDF streams. We consider the case of REST services in Section 5, and we provide an example that shows the use of the DSL with the OWL API in Section 6.

2 Basic RDF Manipulation

A common task in Semantic Web development is basic manipulation of RDF graphs. While sometimes these graphs are loaded as files, it is often needed to build them programmatically. Java libraries exist for that purpose, such as Jena [2]. For instance a simple triple (e.g. the resource `http://somewhere/JohnSmith` has the name `"John Smith"` through the VCard FN predicate) can be added to a Jena RDF model as follows:

```
String personURI = "http://somewhere/JohnSmith";
Model model = ModelFactory.createDefaultModel();
model.createResource(personURI).addProperty(VCARD.FN,"John Smith");
```

The code basically creates a model to which a resource is added, along with the property (predicate) and object. In Scala, exactly the same code can be written as follows, usign our DSL after using some tricks such as implicits methods and implicit classes:

```
val personURI = "http://somewhere/JohnSmith"
implicit val model = createDefaultModel
add(personURI,VCARD.FN->"John Smith")
```

We can already see that the triples can be added in a simpler way to the model, while it still uses the underlying Jena library. The `personURI` is implicitly converted into a resource, and the overloaded `add` method takes pairs of predicates and objects. For the interested developer, the definitions of the helper implicit classes and methods can be found in our GitHub repository, along with running versions of all the examples in this paper[1]. While this example seems simple enough, and both the Java and Scala version are almost equivalent in terms of complexity and lines of code, we can notice that only slightly more complex graphs already start to become harder to code, mainly due to boilerplate in the API. For example consider adding the given, family and full name for the previous example:

```
String personURI = "http://somewhere/JohnSmith";
String givenName = "John";
String familyName = "Smith";
String fullName = givenName + " " + familyName;
Model model = ModelFactory.createDefaultModel();
model.createResource(personURI)
     .addProperty(VCARD.FN,fullName)
     .addProperty(VCARD.N, model.createResource()
                               .addProperty(VCARD.Given,givenName)
                               .addProperty(VCARD.Family,familyName));
```

The resulting code is already a bit cumbersome and bulky, while the Scala counterexample is much terser and focuses only on the code that matters, and the nesting of the

[1] Examples code: `https://github.com/jpcik/morph-streams/tree/master/rdf-streams`

triples can be naturally written, almost as if it was Turtle RDF representation. Again, To get here we are using syntax sugar through implicit classes and methods, which result very handy.

```
val personURI = "http://somewhere/JohnSmith"
val givenName = "John"
val familyName = "Smith"
val fullName = s"$givenName $familyName"
implicit val model = createDefaultModel
add(personURI,VCARD.FN->fullName,
              VCARD.N ->add(bnode,VCARD.Given-> givenName,
                                    VCARD.Family->familyName))
```

Another typical task when dealing with RDF data pogramatically, is to iterate over collections of triples or graphs. Functional programming, which is at the core of the Scala language, is well known to provide powerful and scalable abstractions for dealing with collections. We can see in the following example a Jena code snippet where we iterate over RDF nodes and depending on its type we perform some operation.

```
ArrayList<String> names=new ArrayList<String>();
NodeIterator iter=model.listObjectsOfProperty(VCARD.N);
  while (iter.hasNext()){
  RDFNode obj=iter.next();
  if (obj.isResource())
    names.add(obj.asResource().getProperty(VCARD.Family)
                              .getObject().toString());
  else if (obj.isLiteral())
    names.add(obj.asLiteral().getString());
}
```

The task is extremely simple but the resulting code is confusing to say the least, given the procedural way of dealing with collections, and added to that, the non-intuitive type conditions. The equivalent Scala code, again, using our simplified DSL extension makes use of case matching clauses, which naturally lead to different evaluation paths based on types. For collection processing, the built-in map, filter, foldr and foldl functions in Scala greatly simplify the code, making it less error prone and simpler to maintain.

```
val names=model.listObjectsOfProperty(VCARD.N).map{
  case r:Resource=> r.getProperty(VCARD.Family).obj.toString
  case l:Literal=>  l.getString
}
```

3 Querying with SPARQL

Another every-day job in the life of a Semantic Web developer is that of querying RDF datasets, either locally or remotely. The following example makes use of our Scala DSL to query DBpedia. The query is simply written as a multiline string, and the Jena Query is created with the `sparql` object. The `serviceSelect` method executes the query and returns an iterable object of query solutions. In the example we simply get the URI of each resource solution.

```
val queryStr = """select distinct ?Concept
                  where {[] a ?Concept} LIMIT 10"""
val query = sparql(queryStr)
query.serviceSelect("http://dbpedia.org/sparql").foreach{implicit qs=>
  println(res("Concept").getURI)
}
```

For brevity, we do not show the equivalent Java code, but the reader can notice the simplified iteration pattern and convenient use of implicits to avoid code repetition. We expand this example with the use of *Futures*. A future is an object that holds a value that may be available at some point in time[2]. Futures allow creating asynchronous computations that can be made available in the future, providing a powerful abstraction for concurrent programming. We exemplify this, by simply adding the *Future* construct to the `serviceSelect` method of the previous example. The resulting type is now a *future* that encapsulates the iterable object of solutions. In other words this means that this call is asynchronous, and therefore it will not block the lines of code that follow. This is a very useful tool for invocations that can take a long time (e.g. remote SPARQL endpoints), but can be used in other async programming use cases.

```
val f=Future(query.serviceSelect("http://dbpedia.org/sparqll"))
            .fallbackTo(
      Future(query.serviceSelect("http://dbpedia.org/sparql")))
f.recover{ case e=> println("Error "+e.getMessage)}
f.map(_.foreach{implicit qs=>
    println(res("Concept").getValue)
})
```

This example also showcases the built-in resiliency features of futures. Using the `fallbackTo` method of a future, we can call a different SPARQL endpoint if the first one fails (e.g. if it was unavailable). Also, using the `recover` method we can perform eror handling or other cleanup tasks if all fallback options fail.

4 Streams of RDF

RDF data is increasingly made available in very dynamic forms, such as in social networks, sensor data, or the Internet of Things, needing to deal with streams of RDF[3]. In the following example we introduce the notion of Actors, lightweight objects that interact exclusively through a message passing asynchronous model. The example illustrates a simple RDF receiver actor that simply prints a triple if the predicate is `rdf:type`.

[2] Scala Futures: `http://docs.scala-lang.org/overviews/core/futures.html`
[3] W3C Group on RDF Stream Processing: `http://www.w3.org/community/rsp/`

```
class RdfConsumer extends Actor{
   def receive= {
     case t:Triple =>
       if (t.predicateMatches(RDF.‘type‘))
         println(s"received triple $t")
    }
```

The core of the behavior of an actor is defined in the implementation of the `receive` method, and it handles messages (e.g. triples) that arrive to the actor through an internal message box. In this way, concurrent actors can interchange information asynchronously, in local or remote communication. To complete the example, we provide a piece of code that sends triples to this actor in a streaming fashion. The Jena API includes a `StreamRDF` interface that can be implemented to handle sequences of triples as a stream, and we use the send method (`"!"`) to dispatch triples to the actor when they are available in the stream.

```
  val sys=ActorSystem.create("system")
  val consumer=sys.actorOf(Props[RdfConsumer])
  class Streamer extends StreamRDF{
    override def triple(triple:Triple){
      consumer ! triple
    }
  }
```

5 Services: Linked Data Platform

RDF and Linked Data are increasingly made available through different sorts of web services. In the example below we show how we can easily wire up a REST service for reading and writing containers in the style of the Linked Data Platform[4]. For this we use the Play Framework[5] which is available for both Scala and Java. First we configure the routes, defining the verbs and the relative URLs. In the example below the `retrieve` method will be called for the GET verb while the `add` method will be invoked in case of a POST. The `containerid` is a variable in the URL route.

```
GET /containers/:containerid/
    org.rsp.ldp.ContainerApp.retrieve(containerid:String)
POST /containers/:containerid/
    org.rsp.ldp.ContainerApp.add(containerid:String)
```

For example the `retrieve` method can be implemented as follows. It lists all statements whose subject matches the incoming id, and outputs them in Turtle format. Notice that the service calls itself is asynchronous.

[4] Linked Data Platform: `http://www.w3.org/TR/ldp/`
[5] Play Framework web development platform: `http://www.playframework.com/`

```
object ContainerApp extends Controller {
  def retrieve(id:String) = Action.async {implicit request=>
    val sw=new StringWriter
    val statements=m.listStatements(iri(prefix+id+"/"),null,null)
    val model=createDefaultModel
    statements.foreach(i=>model.add(i))
    model.write(sw, "TURTLE")
    Future(Ok(sw.toString).as("text/turtle").withHeaders(
      ETAG->tag,
      ALLOW->"GET,POST" ))
}
```

6 Reasoning and the OWL API

As a final example, we show our DSL extensions for the well known OWL API [1], which is commonly used to deal with OWL ontologies, their axioms and reasoning tasks. The example shows how we can easily declare OWL classes using the `clazz` object, and individuals using the `ind` construct. Moreover we have defined convenient shortcuts for declaring axioms, e.g. `subClassOf`, which greatly simplify this task compared to the original unmodified OWL API. The example is self explanatory. We declare that singer is a sub-class of artist, and that Elvis is a singer. Then after applying reasoning, we list all individuals that are artists (and Elvis should be classified as such).

```
val onto=mgr.createOntology
val artist=clazz(pref+"Artist")
val singer=clazz(pref +"Singer")
onto += singer subClassOf artist

val reasoner = new RELReasonerFactory().createReasoner(onto)

val elvis = ind(pref+"Elvis")
reasoner += elvis ofClass singer
reasoner.reclassify
reasoner.getIndividuals(artist) foreach{a=>
  println(a.getRepresentativeElement.getIRI)
}
```

7 Conclusions

In this paper, we have shown how we can profit from the extensibility and scalability of Scala to tweak existing Java libraries in such a way that we reduce code boilerplate and complexity. Through a series of examples we see that every-day tasks can be simplified using our DSL extensions, leading to terser code which is potentially cleaner and less error prone. Moreover, we have shown how we can use other built-in features of Scala for writing reactive and scalable semantic web code. More precisely we have discussed about the use of the Actor model and Futures for asynchronous programming. Although the examples shown are only the tip of the iceberg (compared to the myriad of Semantic Web use cases that we can find), we believe that it provides a good glimpse of how one can profit from these powerful language features, combined with well-tested libraries that are used everyday in many projects. Although the DSL that we propose is minimalistic and only a proof of concept, it shows the potential of using a type-safe extensible programming language to develop a more reactive Semantic Web.

Acknowledgments Supported by the SNSF Nano-Tera OpenSense2 and CCES Osper projects.

References

1. Horridge, M., Bechhofer, S.: The OWL API: A Java API for OWL ontologies. Semantic Web 2(1), 11–21 (2011)
2. McBride, B.: Jena: A semantic web toolkit. IEEE Internet computing 6(6), 55–59 (2002)
3. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Inc (2008)
4. Segaran, T., Evans, C., Taylor, J.: Programming the semantic web. O'Reilly Media, Inc. (2009)