Reasoning with Jena

1 Entailment calculation

1.1 Exercise

Create a program which reads a model, applies RDFS reasoning and outputs only the new entailed triples, and not including the RDFS axiomatic triples, e.g., like
`rdfs:Class rdf:type rdfs:Resource`.

1.1.1 Tip

One way to solve the exercise is to create an RDFS model from the read model, create an empty RDFS model (this model have only the axiomatic RDFS triples), and then remove the triples in the latter model from the former model using `Model.difference`.

1.1.2 Solution

The program starts with the read and write model methods we have seem before.

```
 1  import org.apache.jena.rdf.model.*;
 2  import org.apache.jena.util.*;
 3  public class RDFSEntailmentDiff {
 4
 5    public Model readModel(String file) {
 6      return FileManager.get().loadModel(file);
 7    }
 8    public void writeModel(Model model) {
 9      model.write(System.out, "N3");
10    }
```

This is the interesting method in the program. First we create an RDFS model `inf` of the input model. This model contains all the RDFS inferred triples. Next we create an `empty` RDFS model which only contains the triples that can be inferred from an empty model. These statements are exactly the RDFS axiomatic triples and the entailments from these triples which are the

ones that we do not want in the output. Then we create the model to be output, i.e., the `inf` model without the statements in the `empty` model and the statements in the input `model`. Finally, we add the same prefixes to the output model as the input model had so that the written output is easier to digest.

```
11  public Model getEntailmentDiff(Model model){
12    InfModel inf = ModelFactory.createRDFSModel(model);
13    InfModel empty =
14      ModelFactory.createRDFSModel(ModelFactory.createDefaultModel());
15    Model entails = inf.difference(empty).difference(model);
16    entails.setNsPrefixes(model);
17    return entails;
18  }
```

`main`.

```
19    public static void main(String[] args){
20      RDFSEntailmentDiff dave = new RDFSEntailmentDiff();
21      Model model = dave.readModel(args[0]);
22      dave.writeModel(dave.getEntailmentDiff(model));
23    }
24  } //end RDFSEntailmentDiff
```

1.2 Exercise

Use your program to find the RDFS inferred triples from the Animal RDF graph in the exercises week 3. Use the graph given in the solution of this exercise.

1.2.1 Result

```
@prefix :        <http://www.example.org#> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .

:Fish
      a       rdfs:Resource , rdfs:Class ;
      rdfs:subClassOf :Fish .

:Bear
      a       rdfs:Resource , rdfs:Class ;
      rdfs:subClassOf :Bear , :Animal .

:Mammal
      a       rdfs:Resource , rdfs:Class ;
```

```
      rdfs:subClassOf :Mammal .

:Animal
      a       rdfs:Resource , rdfs:Class ;
      rdfs:subClassOf :Animal .

:Whale
      a       rdfs:Resource , rdfs:Class ;
      rdfs:subClassOf :Whale , :Animal .

:Cat  a       rdfs:Resource , rdfs:Class ;
      rdfs:subClassOf :Cat , :Animal .
```

2 Entailment checker

2.1 Exercise

Write a java program which reads two RDF graphs and checks if the first graph entails the second by RDFS entailment. The program should return true/false if the first graph entails / does not entail the second graph.

Note that you need to consider blank nodes with special care. Explain why. See tip and ponder on the meaning of *(un)sound* and *(in)complete*.

2.1.1 Tip

Blank nodes can be treated differently, either by

1. ignoring them (bad solution: this will make your program logically *unsound* and *incomplete*),
2. outputting an "I don't know" message when appropriate (better: your program will be *sound*, but still *incomplete*),
3. or by the strategy explained below, or an equivalent one (perfect! your program is both sound and complete with respect to RDFS semantics).

The strategy of my program is to "manually" apply the two simple entailment rules, se1 and se2, to the a model containing the statements of `entailments.n3`, but with the additional restriction that the blank nodes to be added are collected from the statement to be checked for entailment. This extra restriction ensures that the process of adding blank nodes terminates—which is probably the reason why these rules are not included in Jena RDFS reasoning. Then create an RDFS model from this model and check if all the triples in the statement to be checked for entailment is contained in the RDFS model.

You may want to go about solving this exercise in steps, first the bad solution, when the better one, and of course, finish with the perfect one.

Running my program with the `entailments.n3` graph introduced in an earlier exercise as the first graph and

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix : <http://example.org#> .
:Father rdfs:subClassOf :Person .
```

as second graph, gives me the output:

```
true
```

Changing the second graph to

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix : <http://example.org#> .
:Father rdfs:subClassOf [ rdfs:subClassOf :Person ] .
```

gives me:

```
true
```

2.1.2 Solution

The program contains one method for reading a model, two methods which applies the simple graph rules and one method for checking entailment.

```
1  import org.apache.jena.rdf.model.*;
2  import org.apache.jena.util.*;
3  public class RDFSEntailment {
4
5    public Model readModel(String file) {
6      return FileManager.get().loadModel(file);
7    }
```

The following method is probably best explained by pseudo code.

```
for all statements (s p o) in model 2
  if s is blank node and s is not in model 1
    for all statements (s' p' o') in model 1
      add (s p' o') and (s' p' s) to model 1
```

```
  end if
  if o is blank node and o is not in model 1
    for all statements (s' p' o') in model 1
      add (s' p' o) and (o p' o') to model 1
  end if
end for
```

We need to check if the blank nodes from model 2 are not already present in model 1 as we by omitting this test could create new bindings to the already present nodes in model 1. Consider the example from slide 54 in lecture 2: "Ernest probably did not shoot the female lion he loves."

```
 8  public Model addBNodes(Model m1, Model m2){
 9    StmtIterator stmts2 = m2.listStatements();
10    while(stmts2.hasNext()){
11      Statement stmt2 = stmts2.next();
12      Resource sub2 = stmt2.getSubject();
13      if(sub2.isAnon() && !(m1.containsResource(sub2))){
14   m1 = addBNode(m1, sub2);
15      }
16      RDFNode obj2 = stmt2.getObject();
17      if(obj2.isAnon() && !(m1.containsResource(obj2))){
18   m1 = addBNode(m1, (Resource)obj2);
19      }
20    }
21    return m1;
22  }
```

The following method adds new statements to the argument `model` The statements are collected from `model`, but the subject and object is replaced with the `bnode` argument. In pseudo code (slightly rewritten part of the pseudo code above):

```
for all statements (s p o) in model
  add (bnode p o) and (s p bnode) to model
```

```
23  protected Model addBNode(Model model, Resource bnode){
24    StmtIterator stmts = model.listStatements();
25    Model bNodes = ModelFactory.createDefaultModel();
26    while(stmts.hasNext()){
27      Statement stmt = stmts.next();
28      Statement sg1 = bNodes.createStatement(bnode,
            stmt.getPredicate(), stmt.getObject());
29      bNodes.add(sg1);
30      Statement sg2 = bNodes.createStatement(stmt.getSubject(),
```

```
            stmt.getPredicate(), bnode);
31      bNodes.add(sg2);
32    }
33    return model.union(bNodes);
34  }
```

`RDFSentails` takes two models as arguments. First, all blank nodes from model `m2` which are not present in model `m1` are added to `m1`—as explained above. Then we create an inferred RDFS model from `m1`. This model "automatically" contains all RDFS inferred statements. The method `containsAll` *syntactically* checks if all statements in the argument `m2` exists in `infmodel` and returns `true` or `false` accordingly.

```
35  public boolean RDFSentails(Model m1, Model m2){
36    m1 = addBNodes(m1, m2);
37    InfModel infmodel = ModelFactory.createRDFSModel(m1);
38    return infmodel.containsAll(m2);
39  }
```

`main` reads two models and sends them to `RDFSentails` for entailment checking.

```
40    public static void main(String[] args){
41      RDFSEntailment dave = new RDFSEntailment();
42      Model m1 = dave.readModel(args[0]);
43      Model m2 = dave.readModel(args[1]);
44      System.out.println(dave.RDFSentails(m1, m2));
45    }
46  } //end RDFSEntailment
```

2.2 Exercise

Explain why the three proposed blank node strategies are respectively

1. unsound and incomplete,
2. sound and incomplete,
3. sound and complete.

2.3 Exercise

Use your program to check if the answers from your manual entailment calculation from earlier exercises are correct.

2.3.1 Solution

To do this I have written a tiny shell script which loops through all files called `entailments_*.n3`, which are the RDF files to be checked for entailment, and executes the java program.

2.3.2 Result

```
:Father rdfs:subClassOf :Person .
true

:Man rdfs:subClassOf :Person .
true

:Carl a :Person .
true

:Carl a :Parent .
true

:Carl :hasChild :Ann .
false

:Carl a :Man .
true

:Carl a :Father .
true

:Ann a :Child .
false

:Child rdf:type rdfs:Resource .
true

:Ann :isChildOf :Carl .
false

:Ann :hasParent :Carl .
true

:Ann :hasParent _:x .
true
```

```
:Ann :hasParent [ rdf:type :Person ] .
true

:hasFather rdfs:domain :Person .
false

rdfs:range rdf:type rdfs:Resource .
true

:hasFather rdfs:range :Father .
true

:hasFather rdfs:domain [ rdfs:subClassOf :Person ] .
false

:Father rdfs:subClassOf [ rdfs:subClassOf :Person ] .
true
```

2.4 Exercise

Change your entailment checker program to check for OWL entailment, instead of RDFS entailment.

2.4.1 Solution

My program is a class `OWLEntailment` extending `RDFSEntailment`. It adds a new method `OWLentails` which works similar to the method `RDFSentails` in its superclass, first add blank nodes like explained earlier, then create a reasoner and check if all statements in `m2` are contained in the inferred model. The difference from the two entailment checkers is of course that the OWL entailment checker creates an OWL reasoner and not an RDFS reasoner when creating the inferred model.

```
 1  import org.apache.jena.rdf.model.*;
 2  import org.apache.jena.reasoner.*;
 3  import org.apache.jena.ontology.*;
 4
 5  public class OWLEntailment extends RDFSEntailment {
 6
 7    public boolean OWLentails(Model m1, Model m2){
 8      m1 = addBNodes(m1, m2);
 9
10      Reasoner r = ReasonerRegistry.getOWLReasoner();
11      InfModel inf_m1 = ModelFactory.createInfModel(r, m1);
12
```

```
13      OntModelSpec spec = new OntModelSpec(OntModelSpec.OWL_MEM);
14      OntModel ont_m1 = ModelFactory.createOntologyModel(spec, inf_m1);
15
16      return ont_m1.containsAll(m2);
17    }
18
19    public static void main(String[] args){
20      OWLEntailment dave = new OWLEntailment();
21      Model m1 = dave.readModel(args[0]);
22      Model m2 = dave.readModel(args[1]);
23      System.out.println(dave.OWLentails(m1, m2));
24    }
25  } //end class
```

2.5 Exercise

Run your OWL entailment checker on the same input as the as you did with the RDFS entailment checker. Are there any differences?

2.5.1 Tip

You might want to replace all instances of `rdfs:Class` in `entailments.n3` with `owl:Class`. It seems that OWL reasoners do not reason correctly with RDFS ontologies without some preprocessing.

2.5.2 Result

The OWL entailment checker should return `true` whenever the RDFS entailment checker returns `true`—which it does, at least after changing the vocabulary of the `entailments.n3` file to OWL. The differences lie in the domain and range statements, while RDFS lack the rules to conclude statements about range and domain, OWL does not.

```
:Father rdfs:subClassOf :Person .
  RDFS:true
  OWL:true

:Man rdfs:subClassOf :Person .
  RDFS:true
  OWL:true

:Carl a :Person .
  RDFS:true
  OWL:true
```

```
:Carl a :Parent .
  RDFS:true
  OWL:true

:Carl :hasChild :Ann .
  RDFS:false
  OWL:false

:Carl a :Man .
  RDFS:true
  OWL:true

:Carl a :Father .
  RDFS:true
  OWL:true

:Ann a :Child .
  RDFS:false
  OWL:false

:Child rdf:type rdfs:Resource .
  RDFS:true
  OWL:true

:Ann :isChildOf :Carl .
  RDFS:false
  OWL:false

:Ann :hasParent :Carl .
  RDFS:true
  OWL:true

:Ann :hasParent _:x .
  RDFS:true
  OWL:true

:Ann :hasParent [ rdf:type :Person ] .
  RDFS:true
  OWL:true

:hasFather rdfs:domain :Person .
  RDFS:false
  OWL:true
```

```
rdfs:range rdf:type rdfs:Resource .
  RDFS:true
  OWL:true

:hasFather rdfs:range :Father .
  RDFS:true
  OWL:true

:hasFather rdfs:domain [ rdfs:subClassOf :Person ] .
  RDFS:false
  OWL:true

:Father rdfs:subClassOf [ rdfs:subClassOf :Person ] .
  RDFS:true
  OWL:true
```

3 RDFS metrics

3.1 Exercise

Make a program which loads an RDF(S) file and outputs

- the number of named `rdfs:Class`-es,
- the number of named `rdfs:Property`-es,
- the number of `rdfs:domain` assertions,
- the number of `rdfs:range` assertions,
- the number of `rdfs:subClassOf` axioms,
- the number of `rdfs:subPropertyOf` axioms,
- optionally, the maximum depth of the subclass hierarchy and
- optionally, the maximum depth of the subproperty hierarchy.

The number of classes and properties should also include classes which are not explicitly declared as an `rdfs:Class` or `rdf:Property` (see Tip below), and not include classes or properties which are part of the RDF or RDFS vocabulary, e.g., `rdf:type` and `rdfs:subClassOf`.

The number of subclass and subproperty axioms should only include the axioms explicitly declared in the input file.

The maximum depth of the subclass hierarchy should count the maximum number of consecutive `rdfs:subClassOf` steps it is possible to make in the model, without stepping to an equivalent class. (If `A` is a subclass of `B`

and `B` is subclass of `A`, then they are equivalent.) This means that you have to watch out for loops in the graph.

3.1.1 Tip

You should be able to make use of the RDF metrics program created in an earlier exercise. You will need to use an RDFS reasoner for some of the problems.

Running your metrics program on the following graph

```
1  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
2  @prefix : <http://example.org/> .
3    :a a rdfs:Class ;
4       rdfs:subClassOf :b .
5    :b a rdfs:Class ;
6       rdfs:subClassOf :c .
```

should give you an output similar to this:

```
Named classes: 3
Named properties: 0
Domain axioms: 0
Range axioms: 0
Subclass axioms: 2
Subproperty axioms: 0
Max. depth of class tree: 2
Max. depth of property tree: 0
```

Note that even though `:c` is not explicitly typed as `rdfs:Class`, the class count returns 3. The depth of the subclass hierarchy is 2, since `:a` is a subclass of `:b`, which is a subclass of `:c`.

Running the following graph through your metrics program

```
 1  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
 2  @prefix : <http://example.org/> .
 3    :a rdfs:subClassOf :b .
 4    :b rdfs:subClassOf :c .
 5    :c rdfs:subClassOf :d .
 6    :d rdfs:subClassOf :b .
 7    :c rdfs:subClassOf :e .
 8    :e rdfs:subClassOf :f .
 9    :f rdfs:subClassOf :g .
10    :g rdfs:subClassOf :e .
11
12    :a :relA :b .
```

```
13    :a :relB :c .
14
15    :relA rdfs:subPropertyOf :relB .
16
17    :relA rdfs:range :b .
18    :relB rdfs:domain :a .
```

should give you results similar to this:

```
Named classes: 7
Named properties: 2
Domain axioms: 1
Range axioms: 1
Subclass axioms: 8
Subproperty axioms: 1
Max. depth of class tree: 2
Max. depth of property tree: 1
```

Note that there are two loops in the subclass hierarchy of this graph, `:b – :c – :d – :b` and `:e – :f – :g – :e`, which can "interfere" with the maximum depth calculation of the subclass hierarchy, if you are not careful.

3.1.2 Solution

My program is called RDFSMetrics and extends RDFMetrics. The variable `modelRDFS` holds the model to be examined.

```
1  import org.apache.jena.rdf.model.*;
2  import org.apache.jena.vocabulary.*;
3  import org.apache.jena.util.iterator.*;
4  public class RDFSMetrics extends RDFMetrics{
5     protected static Model modelRDFS;
```

The "problem" when counting classes and properties is that we want reasoning, since we want to count things that are, e.g., of type `rdfs:Class`, but we do not want all the "RDFS stuff" that comes with every inferred RDFS model. My solution is to create an inferred RDFS model from the input model and an inferred RDFS model from an empty model, and then remove all the statements in the inferred input model which also occur in the empty inferred RDFS model.

Note that we also store the "raw" RDF model by using `super.readModel`. This model will come in handy later.

```
 6  public void readModel(String file){
 7     super.readModel(file);
```

```
 8    InfModel infModel = ModelFactory.createRDFSModel(modelRDF);
 9    InfModel emptyRDFS =
        ModelFactory.createRDFSModel(ModelFactory.createDefaultModel());
10    modelRDFS = infModel.difference(emptyRDFS);
11  }
```

The method `getMaxDepth` takes a Property `p` and a Model `m` and finds the length of a chain of `p`'s in `m`—no loops are allowed. To do this the method loops through all subjects which have the property `p` and and finds max chain length with each of the subjects as stating point, and returns the biggest chain length. To do this it uses `getMaxDepthFromResource`.

```
12  public int getMaxDepth(Property p, Model m){
13    int depth = 0;
14    ResIterator ri = m.listResourcesWithProperty(p);
15    while(ri.hasNext()){
16      depth = Math.max(depth, getMaxDepthFromResouce(ri.next(), p, m));
17    }
18    return depth;
19  }
```

`getMaxDepthFromResource` is a "local" recursive version of the above method. It works like any standard "tree depth" method you may have seen in a data structure course, traversing the graph depth first. The difference is that we have to check for loops in the model. We ignore loops by not visiting nodes that have a `p`-path to the node we are currently in.

Since we have applied RDFS reasoning to the model we will by this avoid, e.g., subclass relation loops between equivalent classes.

```
20  public int getMaxDepthFromResouce(Resource s, Property p, Model m){
21    int depth = 0;
22    NodeIterator i = m.listObjectsOfProperty(s, p);
23    while(i.hasNext()){
24      RDFNode o = i.next();
25      if(o.isResource() && !m.contains((Resource)o, p, s)){
26        depth = Math.max(depth,
            1 + getMaxDepthFromResouce((Resource)o, p, m));
27      }
28    }
29    return depth;
30  }
```

The following method counts and prints the results. Notice that different models are sent to the subclass and subproperty counts than the other counts.

```
31  public void printMetrics(){
32    System.out.println("Named classes: " +
33        countStatements(null, RDF.type, RDFS.Class, modelRDFS));
34    System.out.println("Named properties: " +
35        countStatements(null, RDF.type, RDF.Property, modelRDFS));
36    System.out.println("Domain axioms: " +
37        countStatements(null, RDFS.domain, null, modelRDFS));
38    System.out.println("Range axioms: " +
39        countStatements(null, RDFS.range, null, modelRDFS));
40    System.out.println("Subclass axioms: " +
41        countStatements(null, RDFS.subClassOf, null, modelRDF));
42    System.out.println("Subproperty axioms: " +
43        countStatements(null, RDFS.subPropertyOf, null, modelRDF));
44    System.out.println("Max. depth of class tree: " +
45        getMaxDepth(RDFS.subClassOf, modelRDFS));
46    System.out.println("Max. depth of property tree: " +
47        getMaxDepth(RDFS.subPropertyOf, modelRDFS));
48  }
```

main.

```
49    public static void main(String args[]){
50      RDFSMetrics dave = new RDFSMetrics();
51      dave.readModel(args[0]);
52      dave.printMetrics();
53    }
54  } // end class
```

3.2 Exercise

Find the metrics of your family RDFS file. Are the results as expected? Why / why not?

3.2.1 Solution

The results I get are:

```
Named classes: 5
Named properties: 10
Domain axioms: 4
Range axioms: 7
Subclass axioms: 2
Subproperty axioms: 7
Max. depth of class tree: 2
Max. depth of property tree: 2
```

The perhaps unexpected results is that the class counts returns 7, while there seems only to be five `rdfs:Class`'es in the file. The two remaining classes are `xsd:int` and `xsd:string`. The are also `rdfs:Class`'es since they are objects in range axioms. Quoting ch_{range}:

> The triple
>
> ```
> P rdfs:range C
> ```
>
> states that P is an instance of the class `rdf:Property`, that C is an instance of the class `rdfs:Class` and that the resources denoted by the objects of triples whose predicate is P are instances of the class C.