4. Unit: Reasoners, Rules & Ontology Updates

4.1. Unit: Reasoners & Rules

Exercise 4.1 This exercise is like an experiment in a chemical lab course (but there are no explosions, except from maybe null pointers):

- Run the examples from the lecture that need a reasoner both with activating (i) Pellet (which is an OWL-DL Reasoner that solves most of the exercises), and (ii) Jena's internal Rule-based (OWL-RL?) reasoner.
- do not use the Web Service, but use the jar from command line. (note that calls of the form
  ```
  user@foo> time java -cp ... ...
  ```
  also return the time needed for execution)
- do a kind of a protocol report that compares the results.
- try to explain discrepancies whenever they occur, and also explain how the reasoner solves the exercise if it comes to the same result.

Exercise 4.2 Express the food-wine-example from the "Deductive Databases" lecture in OWL. Experiment with the same use-case as there: show that for a nonvegetarian meal, the drink served with it is alcoholic.

Run it both with pellet, and with the internal Jena reasoner.

(DB-Lecture slides from `https://www.dbis.informatik.uni-goettingen.de/Lectures/`, somewhere around Slide 676).

Exercise 4.3 (Hybrid Rules) Transitivity is a "syntactically second-order property" which can be mapped to its first-order instantiated pattern.

Use a hybrid rule to apply transitivity for

- mon:reachable as transitive closure of the (symmetrically stored) mon:neighbor relationship, and
- mon:transitiveFlowsInto as transitive closure for the mon:flowsInto relationship.
- State a SPARQL query against the result that answers all countries that are reachable from Germany,
- State a SPARQL query that computes the length of the total river network for all rivers that flow directly into some sea.
- Note: transitivity is a recursive/inductive property. This exercise requires additional efficiency (concerning handling recursion in general) and Logic Programming strategies, ask the supervisors if your solution does not work as expected.

4.2. Unit: Ontology Updates

Exercise 4.4 (Close-the-World: Win-Move) For the win-move-game in the lecture, for every node, the out-degree was given explicitly, and the file "winmove-closure.n3" contained additional auxiliary classes.

Goal: write a java class that needs as input only "winmove-axioms.n3" and a graph consisting only of edges and nodes. The class should then automatically define the required auxiliary classes and make each node a member of the appropriate one.

- write an appropriate method closeWorld(URI property, OntClass class) that takes a property and a (domain) class and adds all necessary information to close the property wrt. this class. This method should be called as closeWorld(<foo://bla#edge>, <foo://bla#Node>).

- experiment with different input win-move graphs (e.g. the ones from the DD/SemWeb exercises).
- Optional: if you know some good graph visualization program, visualize the outcome with it.

Exercise 4.5 (Close-the-World: Mondial) Goal: check with Mondial which organizations are subsets of others (e.g., BeNeLux is a subset of the NATO, EU, and UN).

- load mondial-europe into a model.
- state a SPARQL SELECT query that outputs for each organization its name and the number of members.
- derive from this a SPARQL CONSTRUCT query that creates all classes $\exists^{=n}\mathsf{hasMember}.\top$ that describe things with n members that are needed. Make each organization a member of the appropriate class.
- define for each organization a class Omembers, e.g. EUmembers, that contains all is members.
- state a SPARQL query that returns all pairs O_1, O_2 of organizations such that O_1members rdfs:subClassOf O_2members.
- verify your solution by stating a suitable simple SPARQL query against mondial.n3.

Exercise 4.6 (Planning) The "furniture" example of the lecture (Slide 512 ff.) can be seen as a small planning problem:

- consider the input as "furniture truck unloaded, where to place the pieces in the house?"
- There is some furniture given/already placed, and the reasoner concludes for some elements (here: the table) where they have to be placed.
- For the other ones (the three beds), there are different possibilities.
- How can these possibilities be computed (consider that this is not StableModels)?
- Choose one of the possibilities where to put it, update the ontology accordingly, and ask the reasoner for the consequences. If there are still several solutions choose again.
- program it in a general way such that it can be applied to arbitrarily big furniture problems.
- Extend the scenario with three plants Plant(yucca), Plant(bonsai), and Plant(efeu) and a music instrument Instrument(piano). It is known that in the livingroom, there is a plant, and in each the bedroom and the guestroom, there is *either* a plant or a music instrument.
  Where is the third plant placed logically?
  Compute the solutions in this case.

Exercise 4.7 (OWL Ontology to Relational Schema) This is a long-term real-world-exercise to be done in several steps.

Consider the transformation from an ER model to a relational model. An OWL ontology like mondial-meta.n3 contains more information than an ER model. So it is possible to derive a relational schema from it algorithmically.

How to do that? You can use Java/Jena, SPARQL queries, and create appropriate auxiliary classes. Use `mondial-er.n3` from `https://www.dbis.informatik.uni-goettingen.de/RDF2Rel/` additionally. It uses owl:AnnotationProperties to distinguish between different types of classes:

- er:Concrete: the classes for which relational tables should be created.
- er:Abstract: classes for which no tables should be created.
- er:Interface: classes that are not from the main geo domain, but rather general geometrical notions.

The solution should work for general ontologies, not only for the Mondial example.

As a very first step, create inverse-definitions automatically for all properties that do not yet have some and add them to the ontology.

First bigger step: compute all pairs (*class*, *property*) such that instances of an er:Concrete *class* might have *property*. At first sight, *class* $\sqsubseteq$ domain(*property*) looks reasonable. This would not consider that e.g. mountains might have a lastEruption property from their Volcano subclass. What is the correct condition? How to verify it with OWL/SPARQL?

What are the next steps?