Jena
A Semantic Web Framework

Overview:

Jena is a Java framework for writing Semantic Web applications. It features:

An RDF API

- statement centric methods for manipulating an RDF model as a set of RDF triples
- resource centric methods for manipulating an RDF model as a set of resources with properties
- cascading method calls for more convenient programming
- built in support for RDF containers - bag, alt and seq
- enhanced resources - the application can extend the behavior of resources
- integrated parsers and writers for RDF/XML (ARP), N3 and N-TRIPLES
- support for typed literals

ARP - Jena's RDF/XML Parser

ARP aims to be fully compliant with the latest decisions of the RDF Core WG. The Jena2 version is compliant with the RDF Core recommendations. ARP is typically invoked using Jena's read operations, but can also be used standalone.

SPARQL query language

SPARQL is an RDF query language and protocol developed within W3C. Jena provides the ARQ query engine which is a complete implementation of the SPARQL query language. In addition there is Joseki, an implementation of the SPARQL protocol.

Persistence

There are two persistence subsystems for Jena - SDB, which employs a custom SQL schema on a wide variety of databases, both open source and proprietary; and TDB, which is a high-performance system using custom storage. Both provide full SPARQL support through ARQ integration.

Reasoning Subsystem

The Jena2 reasoner subsystem includes a generic rule based inference engine together with configured rule sets for RDFS and for the OWL/Lite subset of OWL Full. These reasoners can be used to construct *inference models* which show the RDF statements entailed by the data being reasoned over. The subsystem is designed to be extensible so that it should be possible to plug a range of external reasoners into Jena, though worked examples of doing so are left to a future release. See the reasoner documentation for more details.

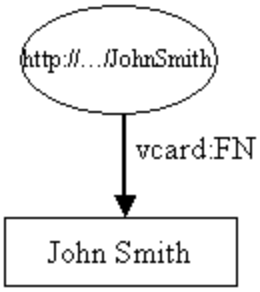
Of these components, the underlying rule engine and the RDFS configuration should be reasonably stable. The OWL configuration is preliminary and still under development.

Ontology Subsystem

The Jena2 ontology API is intended to support programmers working with ontology data based on RDF. Specifically, this means support for OWL, DAML+OIL and RDFS. A set of Java abstractions extend the generic RDF Resource and Property classes to model more directly the class and property expressions found in ontologies using these languages, and the relationships between these classes and properties, and the individuals created from them. The ontology API works closely with the reasoning subsystem to derive additional information that can be inferred from a particular ontology source. Given that ontologists typically modularise ontologies into individual, re-usable components, and publish these on the web, the Jena2 ontology subsystem also includes a document manager that assists with process of managing imported ontology documents.

Introduction

The Resource Description Framework (RDF) is a standard (technically a W3C Recommendation) for describing resources. They use an RDF representation of VCARDS. RDF is best thought of in the form of node and arc diagrams. A simple vcard might look like this in RDF:



The *resource*, John Smith, is shown as an elipse and is identified by a Uniform Resource Identifier (URI)[1], in this case "http://.../JohnSmith.

Resources have *properties*. A property is represented by an arc, labeled with the name of a property. The name of a property is also a URI. The part before the ':' is called a namespace prefix and represents a namespace. The part after the ':' is called a local name and represents a name in that namespace.

Each property has a value. In this case the value is a *literal*, which for now we can think of as a string of characters[2]. Literals are shown in rectangles.

Jena is a Java API which can be used to create and manipulate RDF graphs like this one. Jena has object classes to represent graphs, resources, properties and literals. The interfaces representing resources, properties and literals are called Resource, Property and Literal respectively. In Jena, a graph is called a model and is represented by the Model interface.

The code to create this graph, or model, is simple:

```
// some definitions
static String personURI    = "http://somewhere/JohnSmith";
static String fullName     = "John Smith";

// create an empty Model
Model model = ModelFactory.createDefaultModel ();

// create the resource
Resource johnSmith = model.createResource(personURI);

// add the property
 johnSmith.addProperty(VCARD.FN, fullName);
```

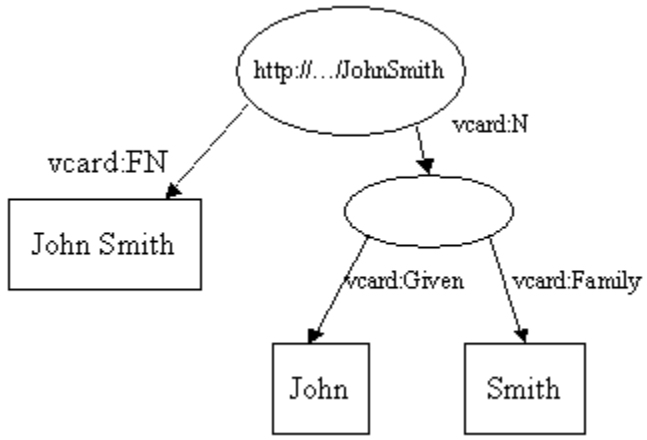
It begins with some constant definitions and then creates an empty Model or model, using the `ModelFactory` method `createDefaultModel()` to create a memory-based model. Jena contains other implementations of the Model interface, e.g one which uses a relational database: these types of Model are also available from ModelFactory.

The John Smith resource is then created and a property added to it. The property is provided by a "constant" class VCARD which holds objects representing all the definitions in the VCARD

schema. Jena provides constant classes for other well known schemas, such as RDF and RDF schema themselves, Dublin Core and DAML.

The code to create the resource and add the property, can be more compactly written in a cascading style:

```
Resource johnSmith =
        model.createResource(personURI)
             .addProperty(VCARD.FN, fullName);
```



Here we have added a new property, vcard:N, to represent the structure of John Smith's name. There are several things of interest about this Model. Note that the vcard:N property takes a resource as its value. Note also that the ellipse representing the compound name has no URI. It is known as an *blank Node.*

The Jena code to construct this example, is again very simple. First some declarations and the creation of the empty model.

```
// some definitions
String personURI    = "http://somewhere/JohnSmith";
String givenName    = "John";
String familyName   = "Smith";
String fullName     = givenName + " " + familyName;

// create an empty Model
Model model = ModelFactory.createDefaultModel();

// create the resource
//   and add the properties cascading style
Resource johnSmith
  = model.createResource(personURI)
         .addProperty(VCARD.FN, fullName)
         .addProperty(VCARD.N,
                      model.createResource()
                           .addProperty(VCARD.Given, givenName)
                           .addProperty(VCARD.Family, familyName));
```

Statements

Each arc in an RDF Model is called a *statement*. Each statement asserts a fact about a resource. A statement has three parts:

- the *subject* is the resource from which the arc leaves
- the *predicate* is the property that labels the arc
- the *object* is the resource or literal pointed to by the arc

A statement is sometimes called a triple, because of its three parts.

An RDF Model is represented as a *set* of statements. Each call of `addProperty` adds an another statement to the Model. (Because a Model is set of statements, adding a duplicate of a statement has no effect.) The Jena model interface defines a `listStatements()` method which returns an `StmtIterator`, a subtype of Java's `Iterator` over all the statements in a Model. `StmtIterator` has a method `nextStatement()` which returns the next statement from the iterator (the same one that `next()` would deliver, already cast to `Statement`). The `Statement` interface provides accessor methods to the subject, predicate and object of a statement.

```
// list the statements in the Model
StmtIterator iter = model.listStatements();

// print out the predicate, subject and object of each statement
while (iter.hasNext()) {
    Statement stmt      = iter.nextStatement();  // get next statement
    Resource  subject   = stmt.getSubject();     // get the subject
    Property  predicate = stmt.getPredicate();   // get the predicate
    RDFNode   object    = stmt.getObject();      // get the object

    System.out.print(subject.toString());
    System.out.print(" " + predicate.toString() + " ");
    if (object instanceof Resource) {
       System.out.print(object.toString());
    } else {
        // object is a literal
        System.out.print(" \"" + object.toString() + "\"");
    }

    System.out.println(" .");
}
```

Since the object of a statement can be either a resource or a literal, the `getObject()` method returns an object typed as `RDFNode`, which is a common superclass of both `Resource` and `Literal`. The underlying object is of the appropriate type, so the code uses `instanceof` to determine which and processes it accordingly.

When run, this program should produce output resembling:

```
http://somewhere/JohnSmith              http://www.w3.org/2001/vcard-rdf/3.0#N
anon:14df86:ecc3dee17b:-7fff .
anon:14df86:ecc3dee17b:-7fff       http://www.w3.org/2001/vcard-rdf/3.0#Family
"Smith" .
anon:14df86:ecc3dee17b:-7fff        http://www.w3.org/2001/vcard-rdf/3.0#Given
"John" .
http://somewhere/JohnSmith   http://www.w3.org/2001/vcard-rdf/3.0#FN      "John
Smith" .
```

Writing RDF

Jena has methods for reading and writing RDF as XML. These can be used to save an RDF model to a file and later read it back in again.

`model.write` can take an `OutputStream` argument.

```
// now write the model in XML form to a file
model.write(System.out);
```

The output should look something like this:

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:vcard='http://www.w3.org/2001/vcard-rdf/3.0#'
 >
  <rdf:Description rdf:about='http://somewhere/JohnSmith'>
    <vcard:FN>John Smith</vcard:FN>
    <vcard:N rdf:nodeID="A0"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="A0">
    <vcard:Given>John</vcard:Given>
    <vcard:Family>Smith</vcard:Family>
  </rdf:Description>
</rdf:RDF>
```

RDF is usually embedded in an <rdf:RDF> element. The element is optional if there are other ways of know that some XML is RDF, but it is usually present. The RDF element defines the two namespaces used in the document. There is then an <rdf:Description> element which describes the resource whose URI is "http://somewhere/JohnSmith". If the rdf:about attribute was missing, this element would represent a blank node.

The <vcard:FN> element describes a property of the resource. The property name is the "FN" in the vcard namespace. RDF converts this to a URI reference by concatenating the URI reference for the namespace prefix and "FN", the local name part of the name. This gives a URI reference of "http://www.w3.org/2001/vcard-rdf/3.0#FN". The value of the property is the literal "John Smith".

Jena has an extensible interface which allows new writers for different serialization languages for RDF to be easily plugged in. The above call invoked the standard 'dumb' writer. Jena also includes a more sophisticated RDF/XML writer which can be invoked by specifying another argument to the `write()` method call:

```
// now write the model in XML form to a file
model.write(System.out, "RDF/XML-ABBREV");
```

This writer, the so called PrettyWriter, takes advantage of features of the RDF/XML abbreviated syntax to write a Model more compactly. It is also able to preserve blank nodes where that is possible. It is however, not suitable for writing very large Models, as its performance is unlikely to be acceptable. To write large files and preserve blank nodes, write in N-Triples format:

```
// now write the model in XML form to a file
model.write(System.out, "N-TRIPLE");
```

Reading RDF

The following code will read it in and write it out. *Note that for this application to run, the input file must be in the current directory.*

```
 // create an empty model
 Model model = ModelFactory.createDefaultModel();

 // use the FileManager to find the input file
 InputStream in = FileManager.get().open( inputFileName );
if (in == null) {
    throw new IllegalArgumentException(
                                "File: " + inputFileName + " not found");
}

// read the RDF/XML file
model.read(in, null);

// write it to standard out
model.write(System.out);
```

The second argument to the `read()` method call is the URI which will be used for resolving relative URI's. As there are no relative URI references in the test file, it is allowed to be empty. When run above program will produce XML output which looks like:

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:vcard='http://www.w3.org/2001/vcard-rdf/3.0#'
 >
  <rdf:Description rdf:nodeID="A0">
    <vcard:Family>Smith</vcard:Family>
    <vcard:Given>John</vcard:Given>
```

```
  </rdf:Description>
  <rdf:Description rdf:about='http://somewhere/JohnSmith/'>
    <vcard:FN>John Smith</vcard:FN>
    <vcard:N rdf:nodeID="A0"/>
  </rdf:Description>
  <rdf:Description rdf:about='http://somewhere/SarahJones/'>
    <vcard:FN>Sarah Jones</vcard:FN>
    <vcard:N rdf:nodeID="A1"/>
  </rdf:Description>
  <rdf:Description rdf:about='http://somewhere/MattJones/'>
    <vcard:FN>Matt Jones</vcard:FN>
    <vcard:N rdf:nodeID="A2"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="A3">
    <vcard:Family>Smith</vcard:Family>
    <vcard:Given>Rebecca</vcard:Given>
  </rdf:Description>
  <rdf:Description rdf:nodeID="A1">
    <vcard:Family>Jones</vcard:Family>
    <vcard:Given>Sarah</vcard:Given>
  </rdf:Description>
  <rdf:Description rdf:nodeID="A2">
    <vcard:Family>Jones</vcard:Family>
    <vcard:Given>Matthew</vcard:Given>
  </rdf:Description>
  <rdf:Description rdf:about='http://somewhere/RebeccaSmith/'>
    <vcard:FN>Becky Smith</vcard:FN>
    <vcard:N rdf:nodeID="A3"/>
  </rdf:Description>
</rdf:RDF>
```

Operations on inference models

For many applications one simply creates a model incorporating some inference step, using the `ModelFactory` methods, and then just works within the standard Jena Model API to access the entailed statements. However, sometimes it is necessary to gain more control over the processing or to access additional reasoner features not available as *virtual* triples.

Validation

The most common reasoner operation which can't be exposed through additional triples in the inference model is that of validation. Typically the ontology languages used with the semantic web allow constraints to be expressed, the validation interface is used to detect when such constraints are violated by some data set.

A simple but typical example is that of datatype ranges in RDFS. RDFS allows us to specify the range of a property as lying within the value space of some datatype. If an RDF statement asserts an object value for that property which lies outside the given value space there is an inconsistency.

To test for inconsistencies with a data set using a reasoner we use the `InfModel.validate()` interface. This performs a global check across the schema and instance data looking for

inconsistencies. The result is a `ValidityReport` object which comprises a simple pass/fail flag (`ValidityReport.isValid()`) together with a list of specific reports (instances of the `ValidityReport.Report` interface) which detail any detected inconsistencies. At a minimum the individual reports should be printable descriptions of the problem but they can also contain an arbitrary reasoner-specific object which can be used to pass additional information which can be used for programmatic handling of the violations.

For example, to check a data set and list any problems one could do something like:

```
Model data = FileManager.get().loadModel(fname);
InfModel infmodel = ModelFactory.createRDFSModel(data);
ValidityReport validity = infmodel.validate();
if (validity.isValid()) {
    System.out.println("OK");
} else {
    System.out.println("Conflicts");
    for (Iterator i = validity.getReports(); i.hasNext(); ) {
        System.out.println(" - " + i.next());
    }
}
```

The file `testing/reasoners/rdfs/dttest2.nt` declares a property `bar` with range `xsd:integer` and attaches a `bar` value to some resource with the value `"25.5"^^xsd:decimal`. If we run the above sample code on this file we see:

Conflicts
- Error (dtRange): Property http://www.hpl.hp.com/semweb/2003/eg#bar has a typed range Datatype[http://www.w3.org/2001/XMLSchema#integer -> class java.math.BigInteger]that is not compatible with 25.5:http://www.w3.org/2001/XMLSchema#decimal

Whereas the file `testing/reasoners/rdfs/dttest3.nt` uses the value "25"^^xsd:decimal instead, which is a valid integer and so passes.

OWL Configuration

This reasoner is accessed using `ModelFactory.createOntologyModel` with the prebuilt `OntModelSpec` `OWL_MEM_RULE_INF` or manually via `ReasonerRegistery.getOWLReasoner()`.

There are no OWL-specific configuration parameters though the reasoner supports the standard control parameters:

Parameter	Values	Description
PROPtraceOn	boolean	If true switches on exhaustive tracing of rule executions to the log4j *info* appender.
PROPderivationLogging	Boolean	If true causes derivation routes to be recorded internally

		so that future getDerivation calls can return useful information.

As we gain experience with the ways in which OWL is used and the capabilities of the rule-based approach we imagine useful subsets of functionality emerging - like that that supported by the RDFS reasoner in the form of the level settings.

OWL Example

As an example of using the OWL inference support, consider the sample schema and data file in the data directory - owlDemoSchema.xml and owlDemoData.xml.

The schema file shows a simple, artificial ontology concerning computers which defines a GamingComputer as a Computer which includes at least one bundle of type GameBundle and a component with the value gamingGraphics.

The data file shows information on serveral hypothetical computer configurations including two different descriptions of the configurations "whiteBoxZX" and "`bigName42`".

We can create an instance of the OWL reasoner, specialized to the demo schema and then apply that to the demo data to obtain an inference model, as follows:

```
    Model schema =
FileManager.get().loadModel("file:data/owlDemoSchema.owl");
    Model data = FileManager.get().loadModel("file:data/owlDemoData.rdf");
    Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
    reasoner = reasoner.bindSchema(schema);
    InfModel infmodel = ModelFactory.createInfModel(reasoner, data);
```

A typical example operation on such a model would be to find out all we know about a specific instance, for example the `nForce` mother board. This can be done using:

```
    Resource nForce = infmodel.getResource("urn:x-hp:eg/nForce");
    System.out.println("nForce *:");
    printStatements(infmodel, nForce, null, null);
```

where `printStatements` is defined by:

```
    public void printStatements(Model m, Resource s, Property p, Resource o)
{
        for (StmtIterator i = m.listStatements(s,p,o); i.hasNext(); ) {
            Statement stmt = i.nextStatement();
            System.out.println(" - " + PrintUtil.print(stmt));
        }
    }
```

This produces the output:

```
nForce *:
 - (eg:nForce rdf:type owl:Thing)
 - (eg:nForce owl:sameAs eg:unknownMB)
 - (eg:nForce owl:sameAs eg:nForce)
 - (eg:nForce rdf:type eg:MotherBoard)
 - (eg:nForce rdf:type rdfs:Resource)
 - (eg:nForce rdf:type a3b24:f7822755ad:-7ffd)
 - (eg:nForce eg:hasGraphics eg:gamingGraphics)
 - (eg:nForce eg:hasComponent eg:gamingGraphics)
```

Note that this includes inferences based on subClass inheritance (being an `eg:MotherBoard` implies it is an `owl:Thing` and an `rdfs:Resource`), property inheritance (`eg:hasComponent eg:gameGraphics` derives from `hasGraphics` being a subProperty of `hasComponent`) and cardinality reasoning (it is the sameAs `eg:unknownMB` because computers are defined to have only one motherboard and the two diffferent descriptions of `whileBoxZX` use these two different terms for the mother board). The anonymous rdf:type statement referencesthe `"hasValue(eg:hasComponent, eg:gamingGraphics)"` restriction mentioned in the definition of GamingComputer.

A second, typical operation is instance recognition. Testing if an individual is an instance of a class expression. In this case the `whileBoxZX` is identifiable as a `GamingComputer` because it is a `Computer`, is explicitly declared as having an appropriate bundle and can be inferred to have a `gamingGraphics` component from the combination of the `nForce` inferences we've already seen and the transitivity of `hasComponent`. We can test this using:

```
    Resource gamingComputer = infmodel.getResource("urn:x-
hp:eg/GamingComputer");
    Resource whiteBox = infmodel.getResource("urn:x-hp:eg/whiteBoxZX");
    if (infmodel.contains(whiteBox, RDF.type, gamingComputer)) {
        System.out.println("White box recognized as gaming computer");
    } else {
        System.out.println("Failed to recognize white box correctly");
    }
```

Which gnerates the output:

```
White box recognized as gaming computer
```

Finally, we can check for inconsistencies within the data by using the validation interface:

```
    ValidityReport validity = infmodel.validate();
    if (validity.isValid()) {
        System.out.println("OK");
    } else {
        System.out.println("Conflicts");
        for (Iterator i = validity.getReports(); i.hasNext(); ) {
            ValidityReport.Report report = (ValidityReport.Report)i.next();
            System.out.println(" - " + report);
        }
```

```
    }
```

Which generates the output:

```
Conflicts
 - Error (conflict): Two individuals both same and different, may be
   due to disjoint classes or functional properties
Culprit = eg:nForce2
Implicated node: eg:bigNameSpecialMB

... + 3 other similar reports
```

This is due to the two records for the `bigName42` configuration referencing two motherboards which are explicitly defined to be different resources and thus violate the FunctionProperty nature of `hasMotherBoard`.

Querying/Reasoning with Jena and SPARQL

Here is a short overview and comparison of RDF querying with SPARQL and Jena which is presented as follows: 1. SPARQL; 2. SPARQL from inside Jena; 3. Explicit and implicit relations when querying with Jena 4.Querying remote SPARQL endpoitns

1. SPARQL

The Simple Protocol and RDF Query Language (SPARQL) is a SQL-like language for querying RDF data. For expressing RDF graphs in the matching part of the query. SPARQL is emerging as the de-facto RDF query language, and is a W3C Recommendation.

For our purposes SPARQL queries could be executed either directly through the SPARQL query panel in Protege or from inside a JAVA application using the specialised Jena library methods. Both approaches are able to handle queries concerning explicit object and property relations but Jena libraries have the advantage of using a reasoner. Thus queries executed using Jena library methods can return results taking in account also the transitive and inferred relations.

Here is an example for querying the http://www.eswc2006.org/technologies/ontology ontology

```
PREFIX ot: <http://www.opentox.org/api/1.1#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT  ?uri
WHERE {
    ?uri rdfs:subClassOf <ot:Feature>.
}
```

Each SPARQL request starts with PREFIXes which denote the namespaces used in the query afterwards.The first lines defines namespace prefix, the last two lines use the prefix to express a

RDF graph to be matched. Identifiers beginning with question mark ? identify variables. In this query, we are looking for resource ?uri participating in triples with predicates rdfs:subClassOf and want the subjects of these triples. Detailed SPARQL syntax description can be found here.

Since **SPARQL is not aware of the rdfs semantics** (hence SPARQL is not aware of owl's semantics). SPARQL doesn't understand the subclass assertions (which are processed as any assertion) and it doesn't understand that rdfs:subClassOf is transitive. Executed from inside Protege this example would return as result only the direct descendats of the class ot:Feature and nothing else. If we want to obtain a full list of subclasses of the ot:Feature class we need to do it using Jena Lib methods.

2. SPARQL from inside Jena

Jena supports several different ontology model specifications.

OntModelSpec.OWL_MEM_MICRO_RULE_INF contains Transitive Reasoner which can be used to infer rdfs:subClassOf and rdfs:subPropertyOf. In contrast the model OWL_MEM does not contain such. Thus The same query we've shown above would return the same result as in Protege if we run in from inside Jena using the simpler model and it will return the entire subtree of subclass if we run it with the model using inference. Executing the following example you will be able to see the difference in the resultset using the two different models.

```
public static void main (String args[]) {
String SOURCE = "http://www.opentox.org/api/1.1";
String NS = SOURCE + "#";
//create a model using reasoner
OntModel model1 = ModelFactory.createOntologyModel(
OntModelSpec.OWL_MEM_MICRO_RULE_INF);
//create a model which doesn't use a reasoner
OntModel model2 = ModelFactory.createOntologyModel(
OntModelSpec.OWL_MEM);

        // read the RDF/XML file
        model1.read( SOURCE, "RDF/XML" );
        model2.read( SOURCE, "RDF/XML" );
        //prints out the RDF/XML structure
        qe.close();
        System.out.println(" ");


    // Create a new query
    String queryString =
      "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> "+
        "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>  "+
        "select ?uri "+
        "where { "+
         "?uri rdfs:subClassOf <http://www.opentox.org/api/1.1#Feature>  "+
        "} \n ";
    Query query = QueryFactory.create(queryString);

    System.out.println("----------------------");
```

```
    System.out.println("Query Result Sheet");

    System.out.println("----------------------");

    System.out.println("Direct&Indirect Descendants (model1)");

    System.out.println("-------------------");


    // Execute the query and obtain results
    QueryExecution qe = QueryExecutionFactory.create(query, model1);
    com.hp.hpl.jena.query.ResultSet results =  qe.execSelect();

    // Output query results
    ResultSetFormatter.out(System.out, results, query);

    qe.close();

    System.out.println("----------------------");
    System.out.println("Only Direct Descendants");
    System.out.println("----------------------");

    // Execute the query and obtain results
    qe = QueryExecutionFactory.create(query, model2);
    results =  qe.execSelect();

    // Output query results
    ResultSetFormatter.out(System.out, results, query);
   qe.close();
}
```

Prints out the following result:

```
----------------------
Query Result Sheet
----------------------
Direct&Indirect Descendants (model 1)
-----------------------------------------------------
| uri                                               |
=====================================================
| <http://www.opentox.org/api/1.1#NumericFeature>   |
| <http://www.opentox.org/api/1.1#NominalFeature>   |
| <http://www.opentox.org/api/1.1#StringFeature>    |
| <http://www.opentox.org/api/1.1#Feature>          |
| <http://www.w3.org/2002/07/owl#Nothing>           |
| <http://www.opentox.org/api/1.1#Identifier>       |
| <http://www.opentox.org/api/1.1#ChemicalName>     |
| <http://www.opentox.org/api/1.1#IUPACName>        |
| <http://www.opentox.org/api/1.1#InChI>            |
| <http://www.opentox.org/api/1.1#MolecularFormula> |
| <http://www.opentox.org/api/1.1#CASRN>            |
| <http://www.opentox.org/api/1.1#SMILES>           |
-----------------------------------------------------

Only Direct Descendants (model 2)
```

```
-------------------------------------------------
| uri                                           |
=================================================
| <http://www.opentox.org/api/1.1#NumericFeature> |
| <http://www.opentox.org/api/1.1#NominalFeature> |
| <http://www.opentox.org/api/1.1#StringFeature>  |
-------------------------------------------------
```

3. Explicit and implicit relations when querying with Jena predefined methods only

From inside Jena one could query the RDF knowledge base also without using SPARQL but by predefined predicates. Here if one wants to query for transitive and inferred relations he must use again a model with a reasoner. Most of the methods for selection have a boolean argument specifying whether only direct or indirect relations must be analysed. The same example shown above executed using only Jena methods would look as follows.

```
public static void main (String args[]) {

    // create the base model using reasoner
    String SOURCE = "http://www.opentox.org/api/1.1";
        String NS = SOURCE + "#";
    OntModel base = ModelFactory.createOntologyModel(
OntModelSpec.OWL_MEM_MICRO_RULE_INF);
            base.read( SOURCE, "RDF/XML" );

     System.out.println( "These are all direct&indirect subClasses of
Feature");
       OntClass event = base.getOntClass( NS + "Feature" );

   //listSubClasses(bool direct) - the false stands for direct and
indirect descendants

    for (Iterator<OntClass> i = event.listSubClasses(false); i.hasNext();
) {
      OntClass c = (OntClass) i.next();
      System.out.println( c.getURI() );
    }

    System.out.println( "-------------------------" );
      System.out.println( "These are only the direct subClasses of
Feature");
    System.out.println( "-------------------------" );
    OntClass event = base.getOntClass( NS + "Feature" );

    //listSubClasses(bool direct) - true stands for direct descendants
only

    for (Iterator<OntClass> i = event.listSubClasses(true); i.hasNext();
) {
      OntClass c = (OntClass) i.next();
      System.out.println( c.getURI() );
    }
```

```
}
```

4. Querying remote SPARQL services

Jena provides convenience methods for querying and processing results of a remote SPARQL endpoint, via its **QueryExecutionFactory** class:

```
import com.hp.hpl.jena.query.QueryExecution;
import com.hp.hpl.jena.query.QueryExecutionFactory;
import com.hp.hpl.jena.query.ResultSet;
import com.hp.hpl.jena.query.ResultSetFormatter;

        String          ontology_service          =          "http://ambit.uni-
plovdiv.bg:8080/ontology";
        String endpoint = "otee:Endpoints";
        String endpointsSparql =
        "PREFIX ot:<http://www.opentox.org/api/1.1#>\n"+
        "       PREFIX ota:<http://www.opentox.org/algorithms.owl#>\n"+
        "       PREFIX owl:<http://www.w3.org/2002/07/owl#>\n"+
        "       PREFIX dc:<http://purl.org/dc/elements/1.1/>\n"+
        "       PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>\n"+
        "       PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n"+
        "       PREFIX otee:<http://www.opentox.org/echaEndpoints.owl#>\n"+
        "               select ?url ?title\n"+
        "               where {\n"+
        "               ?url rdfs:subClassOf %s.\n"+
        "               ?url dc:title ?title.\n"+
        "               }\n";

 QueryExecution   x   =   QueryExecutionFactory.sparqlService(ontology_service,
String.format(endpointsSparql,endpoint));
 ResultSet results = x.execSelect();
 ResultSetFormatter.out(System.out, results);
```

Results

```
-----------------------------------------------------------------------------
-----------------------------------------------------------------------------
| url                                                                        |
title                                                                       |
=============================================================================
=============================================================================
|  <http://www.opentox.org/echaEndpoints.owl#PhysicoChemicalEffects>         |
"Physicochemical effects "^^<http://www.w3.org/2001/XMLSchema#string>       |
| <http://www.opentox.org/echaEndpoints.owl#ToxicoKinetics>                  |
"Toxicokinetics "^^<http://www.w3.org/2001/XMLSchema#string>                |
| <http://www.opentox.org/echaEndpoints.owl#EcotoxicEffects>                 |
"Ecotoxic effects"^^<http://www.w3.org/2001/XMLSchema#string>               |
| <http://www.opentox.org/echaEndpoints.owl#HumanHealthEffects>              |
"Human health effects"^^<http://www.w3.org/2001/XMLSchema#string>           |
|   <http://www.opentox.org/echaEndpoints.owl#EnvironmentalFateParameters>   |
"Environmental fate parameters "^^<http://www.w3.org/2001/XMLSchema#string>
```

OWL notes and limitations

Comprehension axioms

A critical implication of our variant of the instance-based approach is that the reasoner does not directly answer queries relating to dynamically introduced class expressions.

For example, given a model containing the RDF assertions corresponding to the two OWL axioms:

```
class A = intersectionOf (minCardinality(P, 1), maxCardinality(P,1))
class B = cardinality(P,1)
```

Then the reasoner can demonstrate that classes A and B are equivalent, in particular that any instance of A is an instance of B and vice versa. However, given a model just containing the first set of assertions you cannot directly query the inference model for the individual triples that make up *cardinality(P,1)*. If the relevant class expressions are not already present in your model then you need to use the list-with-posits mechanism described above, though be warned that such posits start inference afresh each time and can be expensive.

Actually, it would be possible to introduce comprehension axioms for simple cases like this example. We have, so far, chosen not to do so. First, since the OWL/full closure is generally infiite, some limitation on comprehension inferences seems to be useful. Secondly, the typical queries that Jena applications expect to be able to issue would suddenly jump in size and cost - causing a support nightmare. For example, queries such as (a, rdf:type, *) would become near-unusable.

Approximately, 10 of the OWL working group tests for the supported OWL subset currently rely on such comprehension inferences. The shipping version of the Jena rule reasoner passes these tests only after they have been rewritten to avoid the comprehension requirements.

Prototypes

As noted above the current OWL rule set introduces prototypical instances for each defined class. These prototypical instances used to be visible to queries. From release 2.1 they are used internally but should not longer be visible.

Direct/indirect

We noted above that the Jena reasoners support a separation of direct and indirect relations for transitive properties such as subClassOf. The current implementation of the full and mini OWL reasoner fails to do this and the direct forms of the queries will fail. The OWL Micro reasoner, which is but a small extension of RDFS, does suppor the direct queries.

This does not affect querying though the Ontology API, which works around this limitation. It only affects direct RDF accesses to the inference model.

Performance

The OWL reasoners use the rule engines for all inference. The full and mini configurations omit some of the performance tricks employed by the RDFS reasoner (notably the use of the custom transitive reasoner) making those OWL reasoner configurations slower than the RDFS reasoner on pure RDFS data (typically around x3-4 slow down). The OWL Micro reasoner is intended to be as close to RDFS performance while also supporting the core OWL constructs as described earlier.

Once the owl constructs are used then substantial reasoning can be required. The most expensive aspect of the supported constructs is the equality reasoning implied by use of cardinality restrictions and FunctionalProperties. The current rule set implements equality reasoning by identifying all sameAs deductions during the initial forward "prepare" phase. This may require the entire instance dataset to be touched several times searching for occurrences of FunctionalProperties.

Beyond this the rules implementing the OWL constructs can interact in complex ways leading to serious performance overheads for complex ontologies. Characterising the sorts of ontologies and inference problems that are well tackled by this sort of implementation and those best handled by plugging a Description Logic engine, or a saturation theorem prover, into Jena is a topic for future work.

One random hint: explicitly importing the owl.owl definitions causes much duplication of rule use and a substantial slow down - the OWL axioms that the reasoner can handle are already built in and don't need to be redeclared.

Incompleteness

The rule based approach cannot offer a complete solution for OWL/Lite, let alone the OWL/Full fragment corresponding to the OWL/Lite constructs. In addition the current implementation is still under development and may well have ommisions and oversights. We intend that the reasoner should be sound (all inferred triples should be valid) but not complete.

--Project Report Submitted By
Krishna Priyanka Chebrolu
Ranjani Sankaran