ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

LIBRERIA RDF IN AMBIENTE TUPROLOG

Elaborata nel corso di: Linguaggi e modelli computazionali

Autore:
MATTIA OCCHIUTO

Professore:
MIRKO VIROLI

ANNO ACCADEMICO 2010-2011

Indice

1 Introduzione **1**

1.1 Web Semantico 1

1.1.1 Rdf 2

1.1.2 Owl 2

1.2 Requisiti 3

1.3 Analisi e scelta del tool 3

1.4 Jena 4

2 Libreria RdfLibrary **5**

2.1 Tipi di dati 5

2.2 Caricare file rdf/owl da path 8

2.3 Caricare file rdf/owl da url 9

2.4 Eseguire query rdf 9

2.5 Eseguire una query SPARQL 13

2.6 Eseguire una query SPARUL 14

2.7 Inserire triple nel modello 15

2.8 Eliminare triple dal modello 16

2.9 Caricare specifica rdf definita manualmente dall'utente . . . 17

2.10 Caricare uno schema owl/rdfs associato ad un modello istanza rdf 19

2.11 Validità di un modello 20

2.12 Verifica istanza di classe 21

2.13 Stampare su file il contenuto dei modelli caricati 22

3 Esempi di utilizzo **23**

3.1 File data Rdf 23

3.1.1 Esempio 1 23

3.1.2 Esempio 2 24
3.2 File schema owl e model rdf 25
3.2.1 Esempio 3 25

Capitolo 1

Introduzione

L'obiettivo che questo progetto si pone è quello di introdurre in TuProlog la possibilità di analizzare, manipolare ed istanziare file con contenuto semantico dei tipi: rdf, rdfs e owl attraverso predicati build-in. Viene inoltre richiesto che sia possibile eseguire interrogazioni non banali su questi tipi di dati, effettuare verifiche di validità tra modelli e schemi dei modelli, e aggiornare i modelli con l'inserimento o l'eliminazione dinamica di dati.

Tale progetto cerca quindi di produrre una libreria che permetta di integrare TuProlog in un contesto di Web Semantico.

1.1 Web Semantico

Con il termine web semantico, termine coniato dal suo ideatore, Tim Berners-Lee, si intende la trasformazione del World Wide Web in un ambiente dove i documenti pubblicati (pagine HTML, file, immagini, e così via) siano associati ad informazioni e dati (metadati) che ne specifichino il contesto semantico in un formato adatto all'interrogazione, all'interpretazione e, più in generale, all'elaborazione automatica. Con l'interpretazione del contenuto dei documenti che il Web semantico propugna, saranno possibili ricerche molto più evolute delle attuali, basate sulla presenza nel documento di parole chiave, e altre operazioni specialistiche come la costruzione di reti di relazioni e connessioni tra documenti secondo logiche più elaborate del semplice collegamento ipertestuale.

1.1.1 Rdf

Il Resource Description Framework (RDF) è lo strumento base proposto da W3C per la codifica, lo scambio e il riutilizzo di metadati strutturati e consente l'interoperabilità tra applicazioni che si scambiano informazioni sul Web. è costituito da due componenti:

- RDF Model and Syntax: espone la struttura del modello RDF, e descrive una possibile sintassi.
- RDF Schema: espone la sintassi per definire schemi e vocabolari per i metadati.

L'RDF Data Model si basa su tre principi chiave:

- Qualunque cosa può essere identificata da un Universal Resource Identifier (URI).
- The least power: utilizzare il linguaggio meno espressivo per definire qualunque cosa.
- Qualunque cosa può dire qualunque cosa su qualunque cosa.

1.1.2 Owl

L'Ontology Web Language (OWL) è un linguaggio di markup per rappresentare esplicitamente significato e semantica di termini con vocabolari e relazioni tra gli stessi. Esistono varie versioni del linguaggio, che differiscono molto tra di loro. Lo scopo di OWL è descrivere delle basi di conoscenze, effettuare delle deduzioni su di esse e integrarle con i contenuti delle pagine web. Grazie a OWL in futuro sarà possibile, ad esempio, effettuare delle ricerche estremamente complesse nel web evitando i problemi di omonimia e ambiguità presenti nelle normali ricerche testuali. Altro scopo di OWL è permettere alle applicazioni di effettuare delle deduzioni sui dati. La rappresentazione dei termini e delle relative relazioni è chiamata ontologia. Insieme a RDF, di cui è un'estensione, OWL fa parte del progetto, ancora in itinere, del Web semantico.

1.2 Requisiti

Sono stati specificati, in fase di analisi, i seguenti requisiti :

- caricare file rdf,rdfs e owl da specifica uri o url;
- eseguire semplici query di esistenza su file rdf;
- assocciare un file istanza rdf ad uno modello owl o rdfs;
- verificare la validità del modello caricato nei confronti dello schema associato;
- eseguire query istanza su un file dati rdf rispetto ad uno schema specificato come rdfs o owl;
- utilizzare il linguaggio SPARQL al fine di eseguire query non banali sul modello caricato;
- utilizzare il linguaggio SPARUL al fine di permettere l'aggiunta o la rimozione di triple dal modello caricato;
- permettere di definire e caricare nel modello specifiche definite manualmente.

1.3 Analisi e scelta del tool

E' stato scelto di soddisfare tali requisiti attraverso la produzione di una librearia per l'IDE TuProlog. Tale libreria, che chiameremo RdfLibrary, dovrà proporre abbastanza predicati da rispettare tutte le specifiche del sistema sopra elencate e, qual'ora ne sarà avvertita la necessità, ne implementerà ulteriori. Sempre in questa fase è stato possibile valutare come i requisiti di tale sistema fossero già stati implementati da numerosi tool disponibili.

La scelta fatta è stata quindi quella di considerare le potenzialità proposte da tali tool al fine di poter avere una riusabilità massima di essi e di integrarli quanto più possibile all'interno della RdfLibrary.

Sono stati valutati escluvimente sistemi che fossero di facile integrazione ad un ambiente Java e che avessero maggior supporto e utilizzo da parte

della comunità scientifica; per tali motivi sono stati analizzati e sperimentati principalmente due tool:

- **Pellet** : Pellet è un reasoner OWL, esso fornisce servizi all'avanguardia di reasoning per ontologie OWL.
- **Jena** : Jena è un framework java creato per costruire applicazioni basate sul web Semantico. Esso fornisce un ambiente di sviluppo per RDF, RDFS e OWL, SPARQL e include un motore di inferenza basato su regole.

Per via delle sue limitate funzionalità, la scelta di Pellet porterebbe a dover ricercare altri strumenti che integrino le ulteriori funzionalità richieste. Vista tale considerazione e la completa disponibilità dei servizi richiesti da parte di Jena, si è scelto di utilizzare quest'ultimo come tool sul quale fare affidamento per sviluppare la RdfLibrary.

1.4 Jena

Jena è un framework java per il web semantico open source. Esso fornisce un insieme di API per estrarre e scrivere dati in un grafo RDF. I grafi sono rappresentati come un modello astratto. Un modello può essere caricato con dati provenienti da file, database, URL o combinazioni di questi. Un modello può inoltre essere interrogato attraverso il linguaggio SPARQL o aggiornato con SPARUL.

Capitolo 2

Libreria RdfLibrary

Descriveremo in seguito tutti i singoli predicati introdotti con la libreria RdfLibrary, verranno aggiunte inoltre alcune considerazioni inerenti alla loro relazionalità, ai tipi di parametri utilizzati (input/output) e alle parti di codice salienti utilizzate per la loro implementazione.

2.1 Tipi di dati

Prima di passare alla descrizione dei singoli predicati prodotti presenteremo alcune questioni riguardanti la sintassi rdf. Come detto precedentemente le informazioni rdf sono costituite in triple rappresentate da: soggetto predicato oggetto. Questa struttura delle informazioni fa si che le tre parti costituenti una tripla possano avere significato diverso a livello sintattico; ad esempio un oggetto può essere rappresentato da un url o da una semplice stringa. Inoltre va aggiunto che all'interno di ogni file rdf possono essere utilizzati diverse specifiche rdfs, ad ognuna di queste specifiche è assegnato un prefisso (stringa riassuntiva che rappresenta l'url a cui è associata) ed un set di possibili relazioni utilizzabili tra soggetti ed oggetti presenti nella specifica. Ognuna di queste specifiche è rappresentata da un namespace. Ogni qualvolta un file modello rdf contiene il riferimento a più schemi rdfs e si ha la necessità di riferirsi ad una sua relazione, è necessario poter specificare il namespace relativo.
Per tali motivi è stato necessario aggiungere un concetto di tipo di dato rdf; in questo modo sarà possibile specificare ed interpretare in maniera meno ambigua i predicati ed i risultati prodotti. I tipi vengono distinti attraverso

l'utilizzo di predicati corrispondenti che si differenziano per nome o numero di parametri; quelli inseriti sono tre:

- uri/1
- uri/2
- literal/3

Il primo tipo **uri/1** serve principalmente per contenere soggetti od oggetti di una tripla, essi conterranno valori sotto forma di url o uri.

Il secondo tipo **uri/2** viene utilizzato unicamente per contenere i dati inerenti alle relazioni tra soggetto ed oggetto; nel primo argomento viene specificato il prefisso riferito ad un certo namespace e nel secondo il predicato di relazione (descritto nella specifica riferita dal prefisso) che si vuole utilizzare.

L'ultimo tipo **literal/1** serve a specificare principalmente oggetti delle relazioni e conterrà unicamente dati 'concreti', dove per concreti si intende aventi la forma di semplici stringhe e non di uri o url.

A livello implementativo ciò è stato reso attraverso la classe **TypeTerm**, essa viene utilizzata per eseguire un bind tra il parametro passato ed il tipo che esso rappresenta. Nell'implementazione di tutti i predicati quindi verrà inizialmente istanziata una nuova classe TypeTerm per ogni parametro passato in ingresso, successivamente tale classe conterrà il valore associato al parametro e la descrizione del tipo. In questo modo si è affidato a tale classe l'abilità di gestire la nozione di tipo associata all'informazione, svilcolando la libreria da tali dettagli.
L'implementazione di tale classe ha quindi prodotto:

```
public class TypeTerm {
   ...
    public TypeTerm(Term input){

    //  tipo: 1 = variabile; 2 = url; 3 = url/uri e ns; 4 = literal

    if (input instanceof Var){
       type = 1;
       value = "?"+randomString();
        }
```

```
else if (unify(input, new Struct("uri",valV))){
        type = 2;
        value = ((Struct)valV.getTerm()).getName();
        pred = "uri";
        }
else if (unify(input, new Struct("uri",nsV,valV))){
        type = 3;
        ns = ((Struct)nsV.getTerm()).getName();
        value = ((Struct)valV.getTerm()).getName();
        pred = "uri";
        }
else if (unify(input, new Struct("literal",valV))){
        type = 4;
        value = ((Struct)valV.getTerm()).getName();
        pred = "literal";
        }

}

public String getValueForQ(){

     String result = null;

     switch (type) {
            case 1:
                   result = value;
                   break;

            case 2:
                   result = "<"+value+">";
                   break;

            case 3:
                   result = ns+":"+value;
                   break;

            case 4:
                   result = "\""+value+"\"";
                   break;
```

```
        }
        return result;
    }

    public boolean isVariable(){
        if (type==1) return true;
        else return false;
    }

    public String getPred(){
        return pred;
    }

    public String getValue(){
        return value;
    }

    public String getNs(){
        return ns;
    }

}
```

Nel seguito descriveremo i predicati introdotti, alla luce di quanto qui descritto va quindi detto come i parametri di ingresso verranno sempre passati come uno dei tipi sopra riportati.

2.2 Caricare file rdf/owl da path

predicato : rdf_load_file_1(Term FileName).

Tale operatore ha lo scopo di caricare all'interno di Jena un modello rdf, owl o rdfs specificando il path del file in oggetto; per tale motivo il parametro di ingresso FileName rappresenta il nome del file da caricare intendendo con esso il percorso.

```
private Model model;
....
    public boolean rdf_load_file_1(Term FileName){
    ...
```

```
    model = FileManager.get().loadModel(FName);
    ...
}
```

Viene reso evidente come all'interno della libreria sia presente un istanza della classe Model rappresentante il modello che istanzierà il file attualmente caricato, si evince da ciò che nel caso in cui venisse invocato tale operatore due volte di seguito con argomento due path di file diversi verrebbe considerato come modello attualmente caricato solo quello estratto dal file indicato dall'ultimo path passato.

2.3 Caricare file rdf/owl da url

predicato : rdf_load_url(Term Url).

Capita spesso che i file con specifiche del tipo qui trattato siano localizzati su qualche server nella rete, per tale motivo è stato introdotto un operatore del tutto analogo al precedente tranne che per il significato che ha il parametro passato, il quale in questo caso rappresenta un url.

```
public boolean rdf_load_url_1(Term Url){

     String url = getContString(Url);
     try {
         model.read(url);
         return true;
         }
     ...
}
```

Unico commento da aggiungere rispetto a quanto già detto per l'opertore precedente è far notare come anche in questo caso sia l'istanza model ad essere inizializzata al modello estratto dal file.

2.4 Eseguire query rdf

predicato : query_rdf(Term first, Term sec, Term third).

Con questo predicato è possibile effettuare query sulle triple che costituiscono il modello caricato. Il predicato è costituito da tre argomenti, essi possono essere

utilizzati sia come input che come output specificando o meno un valore; il primo e l'ultimo argomento rappresentano soggetto e oggetto rispettivamente mentre il secondo la relazione che esiste tra i due. Tale predicato è completamente relazionale. Con tale predicato è quindi possibile eseguire qualunque tipo di query a livello di triple rdf inserite nel modello, senza alcun vincolo riguardo alla posizione di variabili di output.

Vediamo ora alcuni dettagli implementativi; in primo luogo va detto che per abilitare tale modalità, non presente nativamente in Jena, è stata utilizzato il supporto fornito dalle query SPARQL. La logica di funzionamento è quindi quella di istanziare dinamicamente una semplice query SPARQL a seconda degli argomenti passati dall'utente attraverso il predicato.

Per fare in modo che alla pressione del tasto next, presente sulla GUI di TuProlog, sia possibile scorrere tutti i risultati trovati con la query, sono state utilizzate le funzionalità ricorsive proprie di prolog.

Come è stato detto precedentemente, in questo caso è stato introdotta una teora prolog utile agli scopi sopra detti, in particolare:

```
public String getTheory() {
     return "query_rdf(A,B,C) :- rdf_query(A,B,C,O),rd(A,B,C,O).\n"
             +"rd(A,B,C,[triple(A,B,C)|T]).                    \n"
             +"rd(A,B,C,[H|T]) :- rd(A,B,C,T).                 \n"

             ...
 }
```

Il predicato che l'utente invoca attraverso il predicato è quindi specificato nella teria qui riportata; nel momento in cui esso viene eseguito la teoria non fa altro che invocare a sua volta il predicato rdf_query/4; quest'ultimo calcola tutti i risultati della query e li inserisce in una lista di predicati triple/3 creata a run-time. Una volta che questa lista è stata creata unificherà con la variabile O presente nel predicato rd/4. A questo punto il predicato rd/4 non farà altro che scorrere tutta la lista di risultati trovati. Come ultimo passaggio tale lista verrà fatta unificare con i parametri di output passati dall'utente, facendola scorrere alla pressione del tasto next della GUI.

```
public boolean rdf_query_4(Term first, Term sec, Term third, Term out){

     Struct finalresult = new Struct();

     TypeTerm Subj = new TypeTerm(first.getTerm());
     TypeTerm Prop = new TypeTerm(sec.getTerm());
```

```
    TypeTerm Obj = new TypeTerm(third.getTerm());

    if(!Prop.isVariable()){
      namespace = model.getNsPrefixURI(Prop.getNs());
      queryString = "PREFIX "+Prop.getNs()+": <"+namespace+"> ";
    }

    queryString = queryString +"SELECT ?x ?y ?z ";

    if(Subj.isVariable()) queryString = queryString+" WHERE {?x ";
    else queryString = queryString+"WHERE { "+Subj.getValueForQ()+" ";

    if(Prop.isVariable()) queryString = queryString+"?y ";
    else queryString = queryString+Prop.getValueForQ()+" ";

    if(Obj.isVariable()) queryString = queryString+"?z}";
    else queryString = queryString+Obj.getValueForQ()+"}";

    Query query = QueryFactory.create(queryString);

    QueryExecution qexec = QueryExecutionFactory.create(query,model);
    ResultSet results = qexec.execSelect();
    ...
}
```

Questa prima parte di codice si occupa della creazione di una query SPARQL, in particolare la variabile string queryString è adibita a contenere tale informazione; la vera e propria esecuzione della query sul modello corrente avviene attraverso l'invocazione del metodo QueryExecutionFactory.create(query,model) nel quale si esplicita l'associazione query-modello. Sempre in questa prima parte di codice si nota come il binding tra termine di input e classe TypeTerm venga effettuato come prima cosa; in questo modo vengono utilizzate le funzionalità proposte da tale classe per la creazione della query.

```
public boolean rdf_query_4(Term first, Term sec, Term third, Term out){

    ...
    while(results.hasNext()){
    QuerySolution result = results.nextSolution();
```

```
    if(Subj.isVariable()){
      sub = (result.get("?x").toString());
      subS = new Struct("uri",new Struct(sub));
    }

    else subS = new Struct(Subj.getPred(),new Struct(Subj.getValue()));

    if(Prop.isVariable()){
      pred = (result.get("?y").toString());
      if(pred.contains("#")){
        String nameSP = model.getNsURIPrefix(pred.split("#")[0]+"#");
        pred = pred.split("#")[1];
        predS = new Struct("uri",new Struct(nameSP),new Struct(pred));
      }else predS = new Struct("uri",new Struct(pred));

    }

    else{ predS = new Struct(Prop.getPred(),new Struct(Prop.getNs()),
                             new Struct(Prop.getValue()));}

    if(Obj.isVariable()){
      obj = (result.get("?z").toString());
      if(obj.contains(":"))
        objS = new Struct("uri",new Struct(obj));
      else
        objS = new Struct("literal",new Struct(obj));

    }

    else { objS = new Struct(Obj.getPred(),new Struct(Obj.getValue()));}

    finalresult.append(new Struct("triple",subS,predS,objS));
    }

    return out.unify(getEngine(), finalresult);
}
```

Nella seconda parte qui proposta quello che viene fatto è scorrere tutte le soluzioni trovate come risultato dell'esecuzione della query e formarle in maniera tale da creare una lista di predicati triple/3 costituiti ad-hoc,essi verranno poi uti-

lizzati nella teoria sopra descritta. Al fine di ritornare il tipo corretto associato ad ogni valore restituito dalla query, prima della creazione del predicato triple vengono creati tanti predicati 'tipo' corrispondenti ad ogni valore presente nelle triple trovate. In questo modo è garantita la coerenza con il tipo che rappresentano come sopra descritto.

2.5 Eseguire una query SPARQL

predicato : sparql_query(Term query, Term Output).

Il predicato sparql_query/2 permette di eseguire query SPARQL sul modello passato; l'analisi di relazionalità di esso è naturale infatti questo lavora tassativamente con il primo argomento come input e il secondo come output. Va fatto notare come il primo argomento dovendo rispettare le regole sintattiche introdotte dal modello SPARQL deve essere una stringa contente la query da effettuare. Anche in questo contesto è naturale aspettarsi numerosi risultati a fronte di una singola query; per tale motivo ,analogamente a quanto già visto per le query rdf, è stata utilizzata una teoria prolog per fare in modo che l'utente potesse scorrere tutta la lista di risultati ottenuti attraverso la pressione del tasto next della GUI di TuProlog.

```
public String getTheory() {
     return ...

            +"query_sparql(A,O) :- sparql_query(A,O1),spa(O,O1). \n"
            +"spa(A,[qRes(A)|T]).                                \n"
            +"spa(A,[H|T]) :- spa(A,T).                          \n"

            ...
    }
```

Il commento a tale teoria è del tutto analoga a quella riportata nella descrizione del predicato query_rdf/3.

Per quanto riguarda il codice utilizzato per interfacciarsi con Jena:

```
public boolean sparql_query_2(Term que, Term out){

     Struct finalresult = new Struct();
     que = que.getTerm();
     String queryString = ((Struct) que).getName();
```

```
    try {
        Query query = QueryFactory.create(queryString);
        QueryExecution qexec = QueryExecutionFactory.create(query,model);
        ResultSet results = qexec.execSelect();

        for (int i=0;results.hasNext();i++){
            QuerySolution result = results.nextSolution();
            finalresult.append(new Struct("qRes",
                                new Struct(result.toString()+"\n")));
        }
    return out.unify(getEngine(), finalresult);
    ...
}
```

Non sono presenti particolari di interesse in tale codice, quello che viene fatto è richiamare le funzionalità proposte da Jena per eseguire query SPARQL sul modello attraverso qexec.execSelect(), successivamente all'aver creato la query da associare al modello esistente QueryExecutionFactory.create(query,model).

2.6 Eseguire una query SPARUL

Attraverso il linguaggio SPARQL è possibile eseguire interrogazioni sul modello rdf caricato; esiste un altro linguaggio, sempre provenienete da SPARQL, che permette di modificare il modello rdf. Tale linguaggio, chiamato SPARUL (SPARQL update), permette di cancelleare ed aggiungere nuove triple al modello esistente.

predicato : sparql_update(Term sQuery)

Tale predicato quindi, in maniera del tutto analoga a sparql_query, permette all'utente di passare, sotto forma di stringa, una query scritta in linguaggio SPARUL e di eseguirla sul modello.

```
public boolean sparql_update_1(Term sQuery) {

    try {
        GraphStore gra = GraphStoreFactory.create(model);
        UpdateRequest request = UpdateFactory.create() ;
        request.add(getContString(sQuery));
```

```
        UpdateAction.execute(request, gra) ;
        return true;
    ...
```

Evidentemente per l'implementazione di tale predicato è stato necessario utilizzare oggetti e classi diverse rispetto all'esecuzione della semplice query SPARQL, ciò nonostante la logica di funzionamento è la medesima. Si crea infatti una query a partire da una stringa e la si esegue sul modello.

I prossimi predicati proposti permetteranno operazioni eseguibili anche con query SPARUL; essi sono stati introdotti al fine di nascondere ad un possibile utente dettagli tecnici associati a tale linguaggio. Si è deciso per questo di inserire due predicati che permettano di: creare e rimuovere triple dal modello.

2.7 Inserire triple nel modello

predicato :
rdf_insert_triple(Term subjT, Term rel, Term objT)

Per capire le problematiche relative alla sintassi SPARUL mostriamo un esempio di esse:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT { <http://example/egbook> dc:title  "This is an example title" }
```

Sono presenti due ulteriori parametri richiesti all'utente oltre ai valori che costituiscono la tripla: il namespace del modello e il prefisso associato ad esso. Fortunatamente all'interno di Jena tali predicati sono messi in relazione in una hasMap, per tale motivo non sono entrambi strettamentene necessari; avendone uno è possibile ottenere facilmente l'altro.

Nell'ambito della libreria qui descritta la specifica del namespace viene eseguita attraverso il secondo parametro; esso infatti è un tipo uri/2, il primo argomento specifica il prefisso al namespace che si desidera utilizzare e il secondo la relazione specificata in tale namespace.

La scelta fatta è stata quella di chiedere all'utente di inserire, come dato aggiuntivo, il prefisso del namespace. Attraverso il prefisso si è infatti in grado di risalire al namespace associato e di completare la query SPARUL.

```
public boolean rdf_insert_triple_3(Term subjT, Term relT, Term objT){

    TypeTerm subjS = new TypeTerm(subjT);
```

```
TypeTerm relS = new TypeTerm(relT);
TypeTerm objS = new TypeTerm(objT);
String namespace = model.getNsPrefixURI(relS.getNs());

String query = "PREFIX "+relS.getNs()+":<"+namespace+"> INSERT
               {"+subjS.getValueForQ()+" "+relS.getValueForQ()+
               " "+objS.getValueForQ()+" } WHERE {}";
try{
   GraphStore gra = GraphStoreFactory.create(model);
   UpdateRequest request = UpdateFactory.create() ;
   request.add(query);
   UpdateAction.execute(request, gra) ;
   return true;
} ...
```

E' possibile verificare come tale implementazione non faccia altro che creare una query SPARUL atta all'inserimento della tripla utilizzando i parametri passati dall'utente; tale implementazione, escluso la parte della creazione della query, è del tutto simile a quanto visto nel predicato precedente.

In ultima analisi va sottolinato come tutti gli argomenti di tale predicato debbano essere necessariamente valori di input e non variabili.

2.8 Eliminare triple dal modello

Nel contesto di questi ultimi due predicati proposti è stato inserito nella libreria anche la possibilità di eliminare triple rdf dal modello. A differenza di quanto detto nel caso dell'inserimento di triple, l'eliminazione nasconde dettagli più complessi. Nel momento in cui si vuole eliminare una tripla infatti, in maniera del tutto analoga a quanto accade per le query di interrogazione, è comprensibile aspettarsi di poter avere come parametro di ingresso una variabile. La presenza di variabili fa si che un utente possa eliminare tutte quelle triple che presentano un certo attributo specifico come soggetto e/o predicato e/o oggetto; in questo modo l'utente ha la possibilità di eliminare un insieme di triple eseguendo un unico predicato.

predicato :
rdf_delete_triple(Term subjT, Term relT, Term objT, Term prefixT)

I parametri associati a tale predicato hanno lo stesso significato e la stessa motivazione spiegata per rdf_insert_triple. Tale predicato, a differenza del precedente, permette però all'utente di inserire variabili al posto dei valori associati

alla tripla rdf. Il range teorico di azione di un predicato di questo tipo va dal cancellare una sola specifica tripla al cancellare tutte le triple presenti nel modello qual'ora tutti i valori della tripla siano variabili.

Per quanto riguarda l'implementazione va detto come la parte più ostica sia quella relativa alla formazione della query.

```
public boolean rdf_delete_triple_3(Term subjT, Term relT, Term objT){

     TypeTerm sub = new TypeTerm(subjT);
     TypeTerm rel = new TypeTerm(relT);
     TypeTerm obj = new TypeTerm(objT);
     String namespace = model.getNsPrefixURI(rel.getNs());

     String query = "PREFIX "+rel.getNs()+": <"+namespace+"> DELETE { "
                    +sub.getValueForQ()+" "+rel.getValueForQ()+" "+
                    obj.getValueForQ()+" }"+" WHERE { "
                    +sub.getValueForQ()+" "+rel.getValueForQ()+" "+
                    obj.getValueForQ()+" }";

     try{
        GraphStore gra = GraphStoreFactory.create(model);
        UpdateRequest request = UpdateFactory.create() ;
        request.add(query);
        UpdateAction.execute(request, gra) ;
        return true;
     } ...
}
```

Come per i precedenti predicati predicati descritti è stato ampiamente fatto uso della classe TypeTerm, e quindi dei tipi associati ai parametri di ingresso.

2.9 Caricare specifica rdf definita manualmente dall'utente

predicato : rdf_load_spec.

Attraverso la libreria RdfLibrary è anche possibile caricare come modello di Jena una specifica scritta manualmente dall'utente come teoria prolog. A questo scopo è essenziale visualizzare prima un esempio di specifica e quali sono

gli operatori permessi/introdotti, successivamente si passerà ad una descrizione degli effetti che tali predicati hanno sul modello e all'implementazione.

```
rdfSpec(createModel(uri(test,'http://myModel#'))).
rdfSpec(rdf_insert_triple(uri('http://resourceUri/mattia'),
        uri(test,'isbrother'),uri('http://resourceUri/steve'))).
rdfSpec(rdf_insert_triple(uri('http://resourceUri/mattia'),
        uri(test,'isbrother'),uri('http://resourceUri/luck'))).
```

Qui è stata riportata un esempio di specifica rdf, essa mostra le caratteristiche minime che deve contenere una specifica di questo tipo. Tutti i predicati che andranno a fare parte della specifica utente sono passati come argomento ad un predicato **rdfSpec**; l'utilizzo di tale predicato equivale quindi ad esplicitare che quanto passato dovrà essere considerato come specifica dell'utente. In primo luogo viene definito il namespace; con la prima specifica **createModel** si crea un nuovo modello all'interno di Jena con associato il prefisso. Tale modello sarà identificato dall'url passato come secondo parametro e avrà associato il prefisso passato nel primo paramatro. Entrambi questi attributi sono a discrezione dell'utente, è però necessario sottolineare che il namespace associato al modello deve avere obbligatoriamente l'aspetto di un url. I predicati successivi **rdf_insert_triple** non fanno altro che aggiungere triple rdf al modello appena istanziato in Jena, qui possono essere utilizzati, per i valori indicanti risorse e relazioni, valori a nostra discrezione è però importante specificare un prefisso corretto che sia presente nel file rdf.

In fase di definizione della specifica è quindi indispensabile rispettare un certo ordine nell'inserimento dei parametri che serviranno ad istanziare la specifica.

Definita la specifica, ciò che resta da fare per far si che essa diventi realmente il modello interno di Jena è invocare il predicato rdf_load_spec.

Verrà descritto in maniera piu' approfondita la logica utilizzata per implementarne le funzionalità in Jena. Per poter analizzare la specifica passata dall'utente ed aggiungerla al modello è necessario utilizzare caratteristiche tipiche di prolog quali la ricorsione; per tale motivo potremmo vedere l'implementazione di questo metodo (e altri) divisa in due parti. In un primo momento sarà infatti necessario analizzare la specifica passata attraverso una teoria prolog; successivamente sarà invece indispensabile utilizzare i metodi forniti da Jena per poter operare concretamente sul modello. Per fare in modo che possano venire specificati fatti prolog e che questi vengano poi asseriti alla teoria è disponibile all'interno della libraria un metodo chiamato getTheory. Tale metodo infatti contiene sotto forma di stringa la teoria che è richiesto venga asserita all'interno della teoria prolog, in questo modo è infatti possibile utilizzare le caratteristiche ricorsive di prolog.

```
public String getTheory() {
     return ...
            +"rdf_load_spec :- rdfSpec(F),!,call(F),retract(rdfSpec(_))
                               ,rdf_load_spec."
            +"rdf_load_spec."
}
```

Tale teoria non fa altro che cercare ricorsivamente nella teoria prolog tutti i fatti rdfSpec e, ogni qualvolta ne trovi uno, invocare il predicato contenuto come argomento di rdfSpec e rimuoverlo dalla teoria attraverso la retract. In tale modo abbiamo la sicurezza che tutti i predicati che rappresentano una specifica vengano asseriti.

Fino a questo momento il modello (vuoto) caricato in Jena non è stato modificato; per fare in modo che le triple fossero caricate all'interno di tale modello e che esso venisse aggiornato correttamente con i valori a queste associati sono state utilizzate query SPARUL. Attraverso tali query infatti è possibile, e di facile implementazione, modificare un modello Jena/Rdf.

Come chiarimento al prossimo predicato presentato e come delucidazione inerente ai tre appena descritti va sottolineato come ogni file caricato in jena venga rappresentato come insieme di triple rdf composte da **soggetto - predicato - oggetto**; ciò fa si che sia possibile mantenere pienamente il contenuto informativo/semantico associato a tale file.

2.10 Caricare uno schema owl/rdfs associato ad un modello istanza rdf

predicato : reasoner(Term modelFile, Term dataFile).

Fino a questo momento abbiamo trattato predicati che operassero principalmente su file modello di tipo rdf; questo operatore invece introduce il concetto di file schema. Attraverso un file di tipo owl/rdfs è possibile definire un insieme di regole associative tra i soggetti che popolano il nostro schema, solitamente (come è stato anche mostrato per l'operatore rdf_load_spec) è necessario specificare quale schema un file rdf rispetta per fare in modo che sia possibile verificare il fatto che tale schema rispetti il modello specificato.

Possiamo dire che un modello (rdf) è istanza di uno schema (owl/rdfs).

In questo contesto, e nel rispetto dei requisiti inizialmente identificati, è stato introdotto tale operatore; esso permette di inserire in Jena modello e schema.

Tale operatore per via della sua natura non gode della completa relazionalità, infatti i parametri sono strettamente necessari al fine del suo funzionamento.

A livello implementativo non si hanno grosse precisazioni da fare, quello che viene fatto è un semplice mapping con i metodi dati da Jena:

```
public boolean reasoner_2(Term modelFile, Term dataFile){

     schema = FileManager.get().loadModel(getContString(modelFile));
     model = FileManager.get().loadModel(getContString(dataFile));
     ...
     if (getContString(modelFile).endsWith(".owl"))
        reasoner = ReasonerRegistry.getOWLReasoner().bindSchema(schema);

     else if (getContString(modelFile).endsWith(".rdfs")){
        reasoner = ReasonerRegistry.getRDFSSimpleReasoner()
                  .bindSchema(schema);

     infmod = ModelFactory.createInfModel(reasoner,model);
     ...
}
```

L'utilizzo di un certo file schema piuttosto che un altro (rdfs/owl sono quelli attualmente utilizzabili) fa si che cambi il tipo di reasoner associato ad essi. Nel predicato successivo vedremo un esempio di come tale reasoner sia indispensabile per eseguire verifichi di validità.

2.11 Validità di un modello

predicato : model_check

Con tale predicato viene utilizzato il reasoner interno a Jena, in particolare tale reasoner è disponibile sia per schemi di tipo rdfs che di tipo owl. L'implementazione di tale predicato è particolarmente banale, non è altresì banale il suo significato; esso infatti permette di verificare se un certo modello è una istanza corretta dello schema a questo associato. Con istanza corretta si intende che il modello deve rispettare pienamente tutte le regole definite nello schema.

```
ValidityReport validity = infmod.validate();
if (validity.isValid())
   return true;
```

```
else {
   for (Iterator i = validity.getReports(); i.hasNext(); ) {
   ValidityReport.Report report = (ValidityReport.Report)i.next();
   getEngine().stdOutput("  -  "  +  report);
}
return false;
```

Va aggiunto come tale predicato non necessiti di alcun argomento, esso infatti lavora implicitamente sui due modelli (di dati e schema) precedentemente caricati in Jena.

2.12 Verifica istanza di classe

predicato : instance_query(Term subjString, Term typeString).

Rimanendo nel contesto introdotto corrente continuiamo l'introduzione di quei predicati che lavorano parallelamente su specifiche schema e modello, in particolare il predicato in esame esegue un controllo di istanza. Tale predicato si occupa di verificare che un certo soggetto/oggetto-istanza appartenente ad un file modello rdf e associato ad un file schema (owl o rdfs) rispetti tale schema e i vincoli da questo imposti.

Esso non è completamente relazionale, sono previsti sostanzialmente due tipi di utilizzi: il primo specificando entrambi i parametri e quindi verificandone la relazione di istanza, il secondo specificando solo l'istanza al fine di ricavare la classe di appartenenza.

In questo contesto è da considerarsi un predicato particolarmente utile sia per assicurarsi che il file rdf sia corretto rispetto allo schema sia per eseguire delle semplici interrogazioni a livello di schema-modello.

A livello implementativo:

```
public boolean instance_query_2(Term subjT, Term typeT){

     TypeTerm subjS = new TypeTerm(subjT);
     TypeTerm typeS = new TypeTerm(typeT);

     Resource subj,type;

     if(!subjS.isVariable() && !typeS.isVariable()){
       subj = infmod.getResource(subjS.getValue());
```

```
      type = infmod.getResource(typeS.getValue());
      return infmod.contains(subj,RDF.type,type);
    }else if(typeS.isVariable()){
      subj = infmod.getResource(subjS.getValue());
      typeT.unify(getEngine(), new Struct("uri",
         new Struct(stringStatements(infmod, subj, RDF.type, null))));
      return true;
    }
    else return false;
}
```

Nella prima condizione verifichiamo che entrambi gli argomenti passati non siano delle variabili e quando questo è rispettato il metodo non fa altro che richiamare funzionalità di Jena, al verificarsi della seconda condizione invece si considera l'eventualità che al posto del secondo argomento istanza venga passata una variabile e a questo punto il sistema si occupa di eseguire una semplice interrogazione con il risultato di unificare il secondo argomento con tutte le possibili istanze riscontrate nel modello.

2.13 Stampare su file il contenuto dei modelli caricati

predicato : printModel(Term fileName).

Quest' ultimo predicato, non presenti nella lista dei requisiti iniziale, è stato inserito per permettere all'utente di avere un riscontro tangibile delle modifiche apportate al modello o del risultato del modello caricato. Esso non fa altro che creare un file avente nome specificato dall'utente contenente la specifica Rdf contenuta nel modello Jena. L'implementazione:

```
public boolean printModel_1(Term fileName){
    String userDir = System.getProperty("user.dir");
    try{
       FileOutputStream out = new FileOutputStream(userDir+"/"+getContString(file
       getEngine().stdOutput(userDir);
       out.close();
       return true;
    }...
}
```

Capitolo 3

Esempi di utilizzo

Verranno mostrati in questo capitolo alcuni esempi pratici di utilizzo della libreria TuProlog. Per entrambi gli esempi va sottolineato come l'IDE messo a disposizione contenga al suo interno la libreria RdfLibrary, essa però non è stata caricata; per tale motivo, in fase preliminare, **è necessario eseguire il 'load' di tale libreria attraverso la finestra Library Manager presente**.

3.1 File data Rdf

In questa prima sezione tratteremo gli operatori che lavorano sui file contenenti dati rdf (non ci occuperemo quindi di schemi rdfs e owl). Verranno elencati in maniera sequenziale gli operatori da eseguire per avere un corretto funzionamento.

3.1.1 Esempio 1

- Carichiamo un modello in Jena :

```
rdf_load_file('code.rdf').
```

- Eseguiamo un'interrogazione sul modello caricato :

```
rdf_insert_triple(uri('http://somewhere/MattiaOcchiuto'),
uri(vcard,'FN'),literal('Mattia Occhiuto')).
```

- Inseriamo una nuova tripla :

```
rdf_delete_triple(X,uri(vcard,'FN'),literal('Sarah Jones')).
```

- Cancelliamo una tripla esistente :

```
rdf_delete_triple(X,uri(vcard,'FN'),literal('Sarah Jones')).
```

- Stampiamo il nuovo modello in un nuovo file :

```
printModel('Esempio1').
```

3.1.2 Esempio 2

- Inserire la specifica come teoria:

```
rdfSpec(createModel(uri(family,'http://myFamily#'))).
rdfSpec(addNameSpace(uri(friend,'http://myFriend#'))).
rdfSpec(rdf_insert_triple(uri('http://resourceUri/mattia'),
        uri(test,'isbrother'),uri('http://resourceUri/steve'))).
rdfSpec(rdf_insert_triple(uri('http://resourceUri/mattia'),
        uri(test,'isbrother'),uri('http://resourceUri/luck'))).
```

- Caricare la specifica passata:

```
rdf_load_spec.
```

- Eseguire una query rdf:

```
query_rdf(uri('http://resourceUri/mattia'),F,G),
```

- Inserire una tripla:

```
rdf_insert_triple(uri('http://resourceUri/mattia'),uri(friend,isfriend),
                  literal(john)).
```

- Eliminare una tripla:

```
rdf_delete_triple(O,uri(friend,isfriend),C).
```

- Stampiamo il modello in un file:

```
printModel('Esempio2').
```

3.2 File schema owl e model rdf

In questa seconda sezione mostreremo un esempio di utilizzo della libreria in riferimento ai predicati che operano sia a livello di modello dei dati che di schema associato.

3.2.1 Esempio 3

- Caricare nel modello un file schema owl e un modello rdf:

```
reasoner('owlDemoSchema.owl','owlDemoData.rdf').
```

- Verificare la validità del modello caricato rispetto allo schema:

```
model_check.
```

- Scoprire la classe di appartenenza di una certa istanza presente nel modello:

```
instance_query(uri('urn:x-hp:eg/bigName42'), C).
```

- Eliminare una tripla dal modello:

```
rdf_delete_triple(uri('urn:x-hp:eg/actionPack'),uri(rdf,type),
                  uri('urn:x-hp:eg/GameBundle')),
```

- Stampare il modello caricato:

```
printModel('Esempio3').
```