Representing and Querying Multiple Ontologies with Contextual Logic Programming

Nuno Lopes[1], Cláudio Fernandes[2], and Salvador Abreu[2]

[1] Digital Enterprise Research Institute, National University of Ireland, Galway
nuno.lopes@deri.org
[2] Universidade de Évora
{cff,spa}@di.uevora.pt

Abstract. The system presented in this paper uses Contextual Logic Programming as a computational hub for representing and reasoning over knowledge modeled by web ontologies, integrating the approach with similar mechanisms which we already developed. The components required to behave as a SPARQL query engine are explained and examples of integration of different sources are shown – in particular, the case of multiple OWL ontologies is discussed.

1 Introduction

The Semantic Web [5] is currently one of the most active and exciting research areas in computer science. Standard web pages provide data adequate for human comprehension, mixing presentation with content, which means that an automated agent cannot easily reason about that information. The Semantic Web is a natural evolution of the World Wide Web which, hopefully, will provide the foundations for intelligent systems and agent layers over the Web.

One important step towards the fulfilling of this vision is the emergence of systems that cannot only understand and reason over Semantic Web documents but also retrieve and process knowledge originating from multiple heterogeneous information sources. One motivation and purpose for our work is to use contextual constraint logic programming [1] as a framework for Semantic Web agents, in which knowledge representation and reasoning for ontology documents can be carried out. As such, we adopted the GNU Prolog/CX programming system, partly described in [2] which incorporates persistence and program structuring through the use of contexts [1]. Throughout this paper, we describe a prototype implementation of a Semantic Web system with three main components:

– A core that is capable of representing web ontologies, integrating that representation with normal GNU Prolog/CX predicates and modules, in order to make them interoperable.
– A SPARQL agent which can answer SPARQL queries about ontologies,
– A back-end capable of mapping GNU Prolog/CX to SPARQL queries, thereby able to query external Semantic Web agents, returning the results as bindings for logic variables present in a GNU Prolog/CX program. This back-end is meant to be integrated with the ISCO [2] framework.

Web Ontology Languages: The Semantic Web is based on existing standard technologies such as XML, RDF and RDF-Schema [12]. Although RDF Schema provides additional modeling primitives, like classes and properties, that enable the hierarchical organization of Web documents, a richer ontology modeling language was necessary. DAML-OIL [8] was then taken as the starting point for the W3C Web Ontology Working Group in defining OWL [13], the language that is aimed to be the standardized and broadly accepted ontology language for the Semantic Web [3]. OWL is defined as an extension of a sub set of the RDF vocabulary and is divided into three species [9]: OWL Lite, OWL DL and OWL Full.

Query Languages: An open research issue has been the specification of a standard query language that can access this kind of data. There are a variety wide of Semantic Web query languages [10], ranging from pure *selection languages* with limited expressivity to general purpose languages supporting different data representation formats and complex queries. Among all the possibilities, we chose to follow the W3C working groups proposed standard: SPARQL [14], an RDF query language and protocol.

The work presented herein is an extension of what was described in [7], we explain some of the implementation choices and introduce some real world examples. The remainder of this article is structured as follows: Contextual Logic Programming is briefly approached in Section 2, in Section 3 we discuss the knowledge representation and ontology querying using Contextual Logic Programming. The issue of querying remote SPARQL agents from within the CxLP framework is discussed in Section 4. Section 5 presents examples of use for the implemented system and Section 6 provides initial conclusions and possible directions for future research.

2 Contextual Logic Programming

Contextual Logic Programming (CxLP) is a simple yet powerful extension to the Prolog logic programming language which provides a mechanism for modularity. In CxLP a finite set of Horn clauses with a given name is designated by *unit*. Abreu and Diaz [1] provide a revised specification for CxLP, which emphasizes the OOP aspects by means of a stateful model, allowed by the introduction of unit arguments. We now informally focus on some aspects of CxLP, namely parametric units; a more complete discussion can be found in [1].

A unit is a parametric module, constituting the program's static definition block. Unit descriptor terms can be instantiated and collected into a list to form a *context*, which can be thought of as a dynamic property of computations. A context specifies the actual *program* (or theory) against which the *current goal* is to be resolved. In short, it specifies the set of predicates which are applicable. These predicates have definitions which result from the specific units which make up the context. A more extensive description of CxLP may be found in [1, 2].

Some parallels can be made between CxLP and Object Oriented Programming (OOP):

Context and object instance: A (possibly partly) bound context is a list of units which can be described as an object *instance*. There is no true analog for the *class* concept, units being conceptually similar to *components*, although the *context term skeleton* may come close.
Predicate and method: A predicate present in a unit is equivalent to a method definition in an OO setting;
Goal and message: a goal executed in a particular context can be interpreted as sending a message (the goal) to an object (the context);
Unit argument and instance variable: unit arguments are variables whose scope is the entire unit, much like instance variables in OO;

GNU Prolog/CX introduces a set of language operators called the *context operators* which are used to modulate the context part of a computation. In a nutshell, when executing a goal G in a context C, a CxLP Engine will traverse C looking for the first unit u that contains a definition for G's predicate. G is then executed as if it were regular Prolog, in a new context that is the suffix of the C which starts with unit u. Some of the most used operations and operators in GNU Prolog/CX are:[1]

Context extension: U :> G, this operation extends the current context with unit U and then reduces goal G in the resulting context;
Context switch: C :< G, evaluates goal G in context C, bypassing the current context;
Supercontext: :^ G, evaluates goal G in the context obtained by removing the top unit from the current context. This is useful to layer specialized on top of generic behavior;
Current context: :< C, unifies C with the current context;
Calling context: :> C, unifies C with the calling context, i.e. the context which was active when the present goal was initially evaluated.

3 System architecture

The initial implementation of the XPTO system is divided in three parts: the core, a SPARQL front-end agent (FE) and a back-end (BE) that maps GNU Prolog/CX to SPARQL queries. The core system is responsible for representing the ontology, the FE enables the resolution of queries expressed in SPARQL and the BE allows the core (and the FE) to query other SPARQL web services. The architecture of XPTO, depicted in Figure 1 (see page 4), is further detailed in [11].

By integrating the core, FE, BE and other Logic Programming frameworks namely ISCO [2], the system is be able to access several heterogeneous sources of information: the ontology, other SPARQL agents or web services, the outcome of Prolog computations as well as relational databases.

[1] For a more detailed and formal description, the reader is referred to [1].

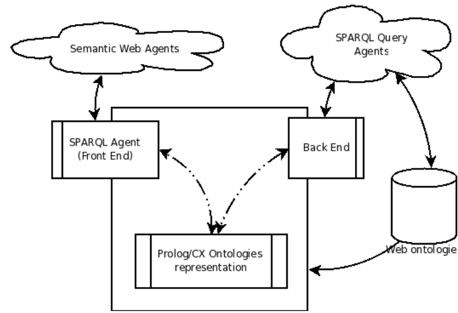

Fig. 1. System architecture

3.1 Representing multiple ontologies

Ontologies are represented using units: there will be one unit that indicates the *elements* (classes and properties) of the ontologies, another unit for *individuals* and one for each OWL *class* and *property*. This arrangement is illustrated in Figure 2.

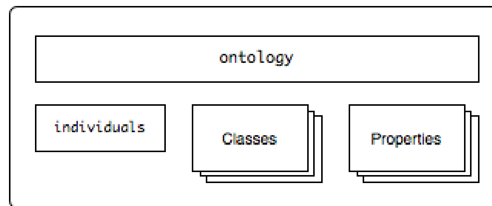

Fig. 2. Ontology representation schema

It is possible to represent several ontologies, i.e. to load more than one ontology into XPTO, and thus querying multiple ontologies also becomes possible. To achieve this, it was necessary to alter both the naming of units and unit files, that is an evolution from the description provided in [11] (where property and class units where named after the property or class). To avoid name clashes, these units were given an internal unique name generated with a prefix `property_` (or `class_`) and a sequential number. A mapping between the unit name and the ontology element is kept in the `ontologies` unit in the predicate that lists the currently available properties and classes: `prop/4` and `class/4` respectively. A simple example follows:

```
prop(ontology_1, 'http://example.org/'#someProp, prop_1).
class(ontology_1, 'http://example.org/'#someClass, class_1).
```

The internal representation of the ontology elements stores the base namespace along with the name of the element, this choice provides a safe behavior when dealing with elements of different ontologies with the same name.

3.2 Querying with multiple ontologies

The new ontology representation schema implies a new query format: it is necessary to explicitly include the namespace of the property or class in the query. This may be a logic variable (as shown in Figure 3) which can be used to either specify or inquire about the namespace part.

```
| ?- N#someClass .>  N#someProp(F) :> item(I).

F = 'http://example.org/'#'somePropValue'
I = 'http://example.org/'#'someIndividual'
N = 'http://example.org/' ?
```

Fig. 3. Query example

4 Mapping Prolog to SPARQL Queries

The back-end component of XPTO behaves as a mapping system from GNU Prolog/CX queries to SPARQL. It is capable of communicating with SPARQL endpoints and therefore enables writing GNU Prolog/CX programs that reason simultaneously over local and external ontologies. Maintaining the query syntax between components of the system is an important objective that will allow programmer to transparently query internal and external ontologies, and other data sources, merging their results in the same program.

To query the external SPARQL endpoints the next steps are followed:

1. The back-end translates a GNU Prolog/CX goal into a SPARQL query;
2. The resulting SPARQL query is sent to the appropriate SPARQL web service;
3. The solution set, described by an XML document, is fetched and the corresponding logic variables which occurred in the original query are nondeterministically bound.

The back-end follows the SPARQL protocol specification [6] which means it will have to provide some pertinent information in order to query an external SPARQL agent. This includes, among others, the `url` of the service, the data format of the response and, possibly, an ontology URI.

4.1 Generating SPARQL SELECT queries

A SPARQL query in the back-end environment is a GNU Prolog/CX context execution similar to the ones defined by the XPTO main mapping engine. The query is always composed of three parts:

1. One URI of the external Semantic Web service;
2. One or more property restrictions;
3. An execution predicate which refers individuals;

Figure 4 illustrates a definition of a back-end query and its three components.

```
QUERY := sparql(URI) /> N1#P1 ... Nn#Pn :> ITEM
         URI  := URL
         P    := property(VALUE) or where(PROP, VALUE)
         ITEM := item(INDIVIDUAL)
```

Fig. 4. Back-End Query Definition

On the left side of the main operator '/>' the external SPARQL endpoint URI is specified and, on the right side, the goals and query restrictions. The latter part of the query triggers the query that will be mapped to SPARQL. This translation is obtained by transforming each property present in the query into a triple that relates the property value to the individual.

The triples are extracted by the union of each **property** term of the right side and the **item** term, which represents the subject of the triple. The following query:

```
sparql('uri.org') />  N1#property1(V1) :>
                      N2#where(PROP, 'value2') :> item(IND).
```

will be translated into the triples:

```
(?IND, N1#property1, ?V1)
(?IND, N2#?PROP, 'value2')
```

All unbound Prolog variables represent SPARQL variables in the triples. To state a value in the query and therefore apply a restriction to the solution, a Prolog atomic value can be used to bind a Prolog variable.

If more than one solution is available for the query, all the results are retrieved using the Prolog backtracking mechanism.

To ask for a property name, i.e, to generate a triple in which the property position is a variable, the auxiliary unit `where/2` should be used (as illustrated

in the previous example). Note that this clause can also be used like a single ask property, by grounding the first argument to a Prolog atom named with some property that describes the individual.

The triples generation process divides the GNU Prolog/CX query information into two parts: the variables, i.e, what is asked and the triple sets. This will result in a direct and transparent translation to SPARQL, where the variables in the query will be the `SELECT` clause arguments and the triples will form the sets in the `WHERE` clause.

4.2 Query example

Figure 5 shows an example of a back-end query that asks a SPARQL endpoint to search the *Wine*[2] ontology for all the individuals that have both `hasBody` and `hasColor` properties.

```
?- sparql('http://xmlarmyknife.org/api/rdf/sparql/') />
     N#hasBody(A) :>
     N#hasColor(B) :> item(I).

A ='Medium'
B ='White'
N ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine'
I ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine' # 'SelaksIceWine'
```

Fig. 5. GNU Prolog/CX query and solution

The generated SPARQL query (Figure 6) is then sent to the SPARQL endpoint. In order to successfully communicate with it, the back-end must first encode the query as specified in the SPARQL Protocol for RDF [6] and if a successful query response code is returned, a solution file is received. This file follows the SPARQL Query Results XML Format [4] and includes one solution. It is then parsed and the solution values are returned as bindings for Prolog variables as illustrated by the last lines in Figure 5.

In the present example, the solution presents exactly one individual, **SelaksIceWine**, and the values **Medium** and **White** for properties *hasBody* and *hasColor* respectively. This means the whole ontology only has one individual that has both of these properties defined.

5 Hybrid Queries

ISCO [2] can be used to integrate, within a Contextual Logic Programming framework, multiple data sources which can then be included in a compound

[2] The ontology is accessible in http://www.w3.org/TR/owl-guide/wine.rdf

```
SELECT ?id ?hasColor ?hasBody
WHERE { ?id :hasColor ?hasColor. ?id :hasBody ?hasBody.}
```

Fig. 6. Generated SPARQL query

Prolog goal. XPTO will include bindings resulting from early goals into later parts of a query to perform joins across multiple sources, some of which may be ontologies queried over SPARQL while others may be, for instance, relational databases.

The code generated by the ISCO framework is static, i.e., it is strictly related to the relational database table, as the code generated by XPTO. The SPARQL code generated is based entirely on the structure of the query: indicating the SPARQL endpoint and, by analysing query patterns, it is possible to determine which ontologies and properties are being queried.

A Prolog goal will generate as many queries as necessary to retrieve the intended information: if all the variables are initially unbound the Prolog engine will traverse all solutions. If any (or all) of the variables are ground or partially instantiated e.g., by using constraints, it is possible to reduce the solution set, as the generated query will already embody these situations.

5.1 Querying Databases and Ontologies

We now present, as an example, queries over a Periodic Table ontology whose individuals are stored in a relational database.

For example purposes, we will use two data sources of information about the periodic table[3]. One will be an ontology that describes the main components of the periodic table like Groups, Blocks and Elements name and the other a database with detailed information about each element. In this case we used an OWL representation of the Periodic Table written by Michael Cook: `http://www.daml.org/2003/01/periodictable/`.

When analysing the definition of a Group in the referred Periodic Table ontology, we can see that each group has, among others, a number, a name and elements. For example, part of group 10 is shown in (Figure 7).

Information about the periodic table elements is present in a database defined with ISCO [2]. Part of the table `element` definition is illustrated in Figure 8 (see page 9).

With the ontology loaded into XPTO system and the database accessible via ISCO, we can write Prolog programs to query over both data sets. Using the `Group` ontology class and the `elements` database table, it is possible to formulate the following query: "classification and color of all the elements belonging to the group `group_10`", as shown in Figure 9.

[3] A periodic table to use as a reference can be found at `http://www.webelements.com/`.

```
<Group rdf:ID="group_10">
    [...]
    <number rdf:datatype="&xsd;integer">10</number>
    <element rdf:resource="#Ni"/>
    <element rdf:resource="#Pd"/>
    <element rdf:resource="#Pt"/>
    <element rdf:resource="#Uun"/>
    [...]
</Group>
```

Fig. 7. Group 10

```
mutable class element.
       code:           int. key.
       name:           text. unique
       symbol:         text. unique
       group:          int.
       color:          text.
       classification: int.
       [...]
```

Fig. 8. Element Table

Variables `ELEMT` and `NUM` will bind together both data sources and, using the Prolog backtrack mechanism, `CLASSF`, `ELEMT` and `COLOR` will return all the solutions available.

6 Conclusion

The system we described and implemented provides a representation abstraction layer for web ontologies that can be accessed by logic programs.

The selected web languages, SPARQL and OWL, have been shown to be appropriate for the scope of our work: to build a working proof-of-concept system which allows us to experiment with Contextual Logic Programming to represent and query ontologies in a way which draws on Prolog's expressiveness as well as the powerful composition mechanisms of CxLP.

We illustrated how this representation can be used to develop Semantic Web agents by describing two components: a front-end and a CxLP back-end. There are aspects of other, existing systems that will benefit from the ability to query SPARQL sources: for instance, we are working on transparently integrating the SPARQL back-end into the ISCO [2] system.

```
| ?-  % access ontology
     'Group' /> N#element(ELEMT) :>
                N#number(_NUM) :> item(group_10),

     % access DB using ISCO
     element@(group=_NUM, name=ELEMT,
              classification=CLASSF, color=COLOR).

CLASSF = 'Metallic'
COLOR = 'lustrous, metallic, silvery tinge'
ELEMT = 'nickel'
N = 'http://www.daml.org/2003/01/periodictable/PeriodicTable'
```

Fig. 9. Query example using ontologies and databases

Future Work Supporting a well-defined OWL sublanguage is necessary in order to provide reliable, trusted semantic web agents which will be usable in wider application sceneries. We are working towards providing provably correct OWL DL compatibility at the reasoning level, over the internal representation. This issue is orthogonal to the rest of the work described herein but it is essential if we expect the system to gain acceptance in the design of Semantic Web agents.

The implemented SPARQL agent currently does not cover the full language specification. Although full SPARQL language support is not our immediate intended purpose, we are working towards providing complete support for it.

A performance study, including a proper comparison with related systems, has already been partially performed [11] and will be further expanded as the implementation evolves.

References

1. Salvador Abreu and Daniel Diaz. Objective: in Minimum Context. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, 2003. ISBN 3-540-20642-6.
2. Salvador Abreu and Vítor Nogueira. Using a Logic Programming Language with Persistence and Contexts. In Masanobu Umeda and Armin Wolf, editors, *Declarative Programming for Knowledge Management*, volume 4369 of *LNCS*, Fukuoka, Japan, 2006. Springer.
3. Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.
4. D. Beckett and J. Broekstra. SPARQL Query Results XML Format. W3C recommendation, W3C, April 2006. Available at: http://www.w3.org/TR/2006/CR-rdf-sparql-XMLres-20060406/.

5. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5), 2001.
6. Kendall Grant Clark. SPARQL Protocol For RDF. Candidate recommendation, World Wide Web Consortium, 6 October 2006. http://www.w3.org/TR/rdf-sparql-protocol/.
7. Nuno Lopes Cláudio Fernandes and Salvador Abreu. On querying ontologies with contextual logic programming. *OWL: Experiences and Directions Third International Workshop*, June 2007.
8. DARPA. DAML. http://www.daml.org/, 3 February 2007.
9. M. Dean, G. Schreiber, S. Bechhofer, Frank van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference. W3C recommendation, W3C, Feb 2004. http://www.w3.org/TR/owl-ref/.
10. Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. Rdf querying: Language constructs and evaluation methods compared. In Pedro Barahona, François Bry, Enrico Franconi, Nicola Henze, and Ulrike Sattler, editors, *Reasoning Web*, volume 4126 of *Lecture Notes in Computer Science*, pages 1–52. Springer, 2006.
11. Nuno Lopes, Cláudio Fernandes, and Salvador Abreu. Contextual logic programming for ontology representation and querying. In Axel Polleres, David Pearce, Stijn Heymans, and Edna Ruckhaus, editors, *2nd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services*, September 2007.
12. Frank Manola and Eric Miller. RDF Primer. W3C Recommendation, World Wide Web Consortium, February 2004.
13. Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, World Wide Web Consortium, February 2004.
14. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2006. Available at: http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/.