On Querying Ontologies with Contextual Logic Programming

Cláudio Fernandes, Nuno Lopes, and Salvador Abreu

Universidade de Évora

Abstract. We describe a system in which Contextual Logic Programming is used as a mediator for knowledge modeled by ontologies. Our system provides the components required to behave as a SPARQL query engine and, as a result of its Logic Programming heritage, it may also recursively interrogate other ontologies or data repositories, providing a semantic integration of multiple sources.

1 Introduction

The Semantic Web topic currently represents one of the most active and exciting research areas in computer science. As the originator and mentor of this vision Tim Berners-Lee puts it [5], the Semantic Web is a natural evolution of the Internet and, hopefully, will provide the foundations for the emergence of intelligent systems and agent layers over the world wide web. The standard web page provides data oriented for human comprehension, which means a computer agent cannot intelligently reason about that information. The Semantic Web thrives the creation of information technology that will allow explicit machine-processable meta-data documents that describes the meaning and semantics of the data published in 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 of multiple information sources. This represents the motivation and purpose of our work which 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 framework Prolog/CX partly described in [2] which makes use of 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, a SPARQL agent which can answer SPARQL queries about ontologies and back-end capable of mapping Prolog/CX to SPARQL queries, thereby able to query external Semantic Web agents, returning the results as bindings for logic variables present in a Prolog/CX program. As available similar work there are other Semantic Web frameworks such as Jena[1] and Sesame.[2]

[1] http://jena.sourceforge.net/
[2] http://www.openrdf.org/

Altough these are more advanced reasoners, interesting points of comparison with our system are at the representation and query level. At this time this is sitll ongoing work.

Web Ontology Languages: Tim Berners-Lee's view and design of the Semantic Web proceeds in steps, each one building a layer on top of the other. At the bottom we find XML, and right above is the RDF and RDF-S [10] layer. RDF is a basic data model for describing simple statements about objects and RDF Schema [10], which is based on RDF, provides additional modeling primitives like classes and properties that enables hierarchical organization of Web documents. However, The Web Ontology Working Group of W3C identified a number of characteristics and use-cases for the Semantic Web that would require more expressiveness than RDF and RDF-S can offer. A richer ontology modeling language was already defined, DAML-OIL [6], which was then taken as the starting point for the W3C Web Ontology Working Group in defining OWL [11], the language that is aimed to, as stated by Grigoris Antoniou and Frank van Harmelen [3] be the standardized and broadly accepted ontology language for the Semantic Web.

Query Languages: There are a variety wide of Semantic Web query languages [9], ranging from pure *selection languages* with limited expressivity to general purpose languages supporting different data representation formats and complex queries. For our research we chose to follow the W3C working groups proposed standard: SPARQL [12], an RDF query language and protocol.

The remainder of this article is structured as follows: after a brief introduction, we introduce contextual logic programming. In section 3 we discuss the issues dealing with knowledge representation and ontology querying from a contextual logic programming perspective. We then proceed, in Section 4, to describe a possible approach to implementing a SPARQL agent, using the CxLP framework. The issue of querying remote SPARQL agents from within the CxLP framework is discussed in Section 5. Finally, Section 6 provides a first experimental evaluation of our system and outline 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. Informally, 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 is applicable. These predicates have definitions which depend on the specific units which make up the context. A more extensive description of CxLP may be found in [1, 2].

GNU Prolog/CX introduces a set of language operators called the *context operators* which 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:[3]

Context extension: U :> G, this operation extends the current context with unit U and then reduces goal G;
Context switch: C :< G, attempts to evaluate goal G in context C, ignoring the current context;
Supercontext: :^ G, evaluates goal G in the context resulting of removing the top unit from the current context;
Current context: :< C, unifies C with the current context;
Calling context: :> C, unifies C with the calling context

3 System architecture

The implemented system is divided in three parts: the core, a front-end (FE) SPARQL agent and a back-end (BE) that maps 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. This architecture is represented in Figure 1. By integrating the core, FE, BE and other Logic Programming frameworks namely ISCO [2], the system will be able to access several heterogeneous sources of information: the ontology, other SPARQL agents or web services and relational data bases.

The main objective of the core system is to represent web ontologies with CxLP tools. After an ontology is transformed into Prolog/CX units, the capabilities of that representation are that of pure Prolog with modular program structuring. For instance, we can build a front end that acts as a SPARQL web agent which can receive a SPARQL query over a known ontology, process it against the internal representation and respond with the solution. This representation can also be used to map Prolog goals to SPARQL queries and collect the results as logic variable bindings. These approaches are further discussed in Sections 4 and 5.

The system we designed is capable of parsing OWL Lite and DL ontologies. Since OWL Full can't guarantee efficient reasoning support, the, OWL DL variant naturally emerged as the goal for the mapping requirements in our system.

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

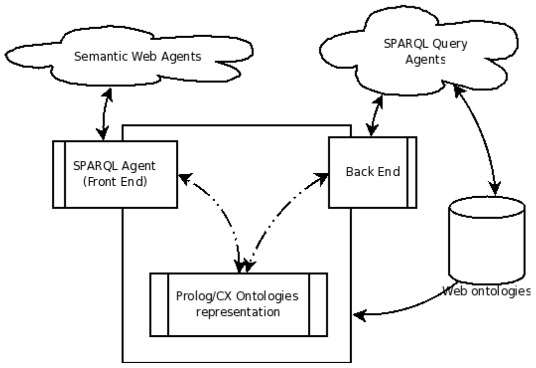

Fig. 1. System architecture

3.1 Ontology representation

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.

The individuals and their property values are represented in the unit `individuals`. This unit stores, for each individual, the class it belongs to and, for each of the individual properties, its value.

Each class and property is defined in a unit named after the class or property. Further information about each of this objects, such as hierarchy and restrictions, can be found in its unit. The naming schema of properties and classes does not pose a problem in OWL DL since, as stated in [7], “OWL DL requires a pairwise separation between classes, datatypes, datatype properties, object properties, annotation properties, ontology properties (i.e., the import and versioning stuff), individuals, data values and the built-in vocabulary”.

The set of known ontologies are represented in a unit named `ontologies` which lists the classes and properties of each loaded ontology. Each property and class listed in this unit can then be accessed in a uniform manner using the operator `/>`. This operator is defined as a context extension operation, i.e., based on the unit name it constructs a new context in which to evaluate the goal.

3.2 Querying an ontology

The most direct way of retrieving the class individuals is to use the goal `item/1`. There is also a goal `item/0` that has the exact behaviour of `item/1` but has no direct arguments, this predicate, when used with the predicate units in the query will allow to access the property values ignoring the name of the individual.

The `item/1` goal binds, by backtrack, its argument to each individual of the class. There is also the possibility of querying all the individuals in the ontology by omitting a class in the query.

The value of the properties can be accessed by including the unit that represents the property in the context query. This enables selecting only a subset of the properties. The argument of the property unit will be bound to the value of the property for the corresponding individual, as shown in Figure 2.

```
| ?- 'IceWine' /> hasFlavor(F) :> hasBody(B) :> item(I).
B = 'Medium'
F = 'Moderate'
I = 'SelaksIceWine' ?
```

Fig. 2. Accessing individuals and properties

3.3 Ontology loading

In order to query an ontology, that ontology must first be transformed and loaded into the system, in a way similar to the compilation process. This results in the ontology being represented as the structure described in Section 3.1.

XML parse: In this step the ontology is handled as a plain XML file and therefore parsed using a standard XML parser.

The selected parser was the "The Expat XML Parser". [4] The parser creates a Prolog term that is an accurate representation of the XML file, and apart from the possible comments in the ontology file, there is no loss of information in this transformation.

Name analysis: The next process is to match the term created by Expat and build a dicnionary with all the information we need to generate the units and predicates that will represent the ontology. The body of the ontology is mapped focusing mostly on classes, properties, individuals and relations between these elements. Ontology headers are also stored to be included in the ontology definition unit.

Unit generation: At this stage, the system has all the information needed to generate the units that will represent the ontology, where each class and property will rise a different unit. Those elements are available in the symbol table, so the mapping engine must walk through it, and for every item generate a unit according with the structure discussed in section 3.1.

[4] http://expat.sourceforge.net/

Compiling and loading the units: In Prolog/CX, an unit must first be compiled and loaded before one can execute its predicates. This means the system, after parsing an ontology and generating the units, must compile and load the Prolog file that contains each unit. This is achieved using the *dynamic loading* of Prolog/CX. Compiling and loading the units represents the last step of the whole core system process, which starts by parsing the ontology, then the name analysis, unit generation, and finally the compilation and loading of the units.

4 A SPARQL agent in CxLP

SPARQL is a Candidate Recommendation for a RDF query language [12]. It is under continued development towards becoming the standard query language for the semantic web [9] and although it is mainly used to query RDF graphs, it can also be used to query an RDF Schema or OWL ontology on the individual and properties level.

The developed system is using SPARQL to query an ontology, allowing access to properties and resulting in individuals and property values and follows the specifications of the language defined in [12] and the results are returned in XML, the format of which is specified in [4]. The parser constructs a Prolog/CX context representing the query; this context is then activated by sending a message to calculate the output and display the resulting XML form. This specification allows our system to be easily made available trough a web service.

SPARQL has 4 types of queries: `select`, `ask`, `construct` and `describe`. The `select` query is used to retrieve the values of the properties and individuals. `Ask` simply returns a boolean answer depending on the veracity of the query. The `construct` and `describe` are not currently implemented as they would return data as RDF graphs.

The following sections briefly describe the SPARQL query language, the resolution of queries and the XML output of the system.

SPARQL and mapping examples The mapping process (SPARQL parser) transforms a SPARQL query into a Prolog/CX context. The execution of this context will bind the variables present in the query with the results.

The context has a similar structure to the SPARQL query, consisting of the following parts, each of which being a parametrized unit:

prefix indicates the default prefix;
from specifies the RDF dataset to query;
select lists the variables that should be present in the output;
where restriction conditions;
Modifiers if present, these modifiers will change the number of results (`limit` and `offset`) and/or their order (`order by`).

The parser receives as input a SPARQL query and returns the context to be executed show in Figure 3.

```
[ all,
  where([set([triple(A,hasFlavor,B),
             triple(A,hasBody,C) ])]),
 select([flavor=B,body=C]),
 vars([flavor=B,body=C,t=A]),
 defs ]
```

Fig. 3. Results of the query example

Query resolution system: The query resolution is triggered by evaluating the goal `item` in the context returned by the mapping process. This is akin to sending the message `item` to an object. The core unit in this process is the unit `triple/1` which is responsible for instantiating the variables in the query by accessing the internal representation of the ontology.

The *Modifiers* will alter the query results, their order or number. If no modifiers are present in the query the unit `all` will be included in the context meaning that all the possible bindings will be returned.

Also, currently, the `from` clause has no effect since the instantiation is done with an already loaded ontology.

XML output: The output of the SPARQL query execution is an XML document with a `sparql` element. This element has then two sub-elements: the `head` element and the `results` element (always shown in this order).

The first element for a `select` query is a list of all the variable names present in the SPARQL query. For an `ask` query (that only returns a boolean) no elements are present. The second element (`results`) is a list of `result` elements. The `result` element has two boolean attributes: ordered and distinct, that are always specified. They indicate, respectively, if the list of results is ordered and if the elements are all different. Its value is defined by the presence or absence of the modifiers `distinct` and `order by` in the SPARQL query.

5 Mapping Prolog to SPARQL Queries

The above described system is a local ontology mapping engine that aims for the creation of a powerful logic layer over ontologies. Access to this system provides a way for reason over local and previous loaded ontologies by means of logic programming. However, that work is meant to be viewed as the foundation of a center system in where it can be appended and implemented different entry and exit points for data access. One possibility is the SPARQL back end presented in this Section. The implemented back end engine aims to transparently merge the reasoning of the system internal knowledge base with external ontologies available from third parties by means of the SPARQL query language. To achieve this, it was developed a back end that provides functions for communicating with web SPARQL agents for ontology querying purposes. It provides the system with the ability to pass a SPARQL query to an arbitrary SPARQL web agent and get the solution, encapsulating the results as bindings for logic variables.

Although the back-end can be viewed as a single independent component of our work, we wanted it to be integrated in a manner that it would allow a programmer to reason over external and internal ontologies using the same query syntax and declarative context mechanics as the main mapping and internal reasoning. Nevertheless, there exists additional information that is needed to be addressed at the external agent such as the `url` of the agent or the data format of the response. `XML Armyknife` agent [8] is used throughout this section to illustrate the back-end functionality. To ensure communication with external agents, the SPARQL protocol for web agents communication [13] was implemented and followed. This W3C *Candidate Recommendation* describes means of conveying SPARQL queries from query clients to a SPARQL query processing service and returning the query results to the requesting entity.

Prolog/CX to SPARQL mapping: A SPARQL query in the back-end environment is a Prolog/CX context execution similar to the ones defined by the main mapping engine when reasoning over a loaded ontology. However, its execution and syntax is slightly different as there is no connection between what is included in the context execution and whatever data is loaded into the main engine of the system. To illustrate how the back-end works, lets introduce the *wine* ontology . [5] This ontology defines classes, properties and individuals and with SPARQL, it is possible to query for RDF triples sets. Figure 4 illustrates a query targeting the `xmlarmyknife.org` [8] SPARQL agent, and consequently the first returned solution.

```
?-sparql('http://xmlarmyknife.org/api/rdf/sparql/') /> hasColor(COLOR):>
hasMaker(MAKER) :> item(I).

COLOR = 'http://www.w3.org/2001/sw/WebOnt/guide-src/wine#White'
I = 'http://www.w3.org/2001/sw/WebOnt/guide-src/wine#SelaksIceWine'
MAKER = 'http://www.w3.org/2001/sw/WebOnt/guide-src/wine#Selaks' ?
```

Fig. 4. Back-end Prolog to SPARQL query

[5] http://www.w3.org/TR/owl-guide/wine.rdf

At the left side of the main operator `"/>"` it is specified the external agent and at the right side the goals and restrictions of the query. The triples are represented by the union of each property term of the right side and the item term, which represents the subject of the triple. This means a context goal like `/> proprety(VALUE) :> item(SUBJ)` represents the triple `(SUBJ, prop, VALUE)`. The above query asks for all the individuals that have the properties `hasColor and hasMaker` and what are their values. This happens because all the arguments of the properties and subject are unbound Prolog variables. To state a value in the query and therefore apply a restriction to the solution, one would write a Prolog atom value instead. Ather possibility is asking for the properties of some individuals, which can be done adding `where(PROP, VALUE)`, where both arguments can be either a Prolog unbound variable or a Prolog atom.

Communication and query solutions: The execution of a back-end query can be described as a three step process. The first is the process of mapping a Prolog/CX query to a SPARQL. The query originates the following SPARQL query: `‘‘SELECT ?id ?hasMaker ?hasColor WHERE  ?id :hasMaker ?hasMaker. ?id :hasColor ?hasColor.’’`.

After generating the SPARQL query, the back-end will start the communication process with the external agent. This includes validating the Web service, sending the query and receiving the response, according to the W3C SPARQL Protocol for RDF document [13]. This can be considered the second step of the back-end query execution. The third and final step consists of parsing the response and retrieving the solution to the query. What is received from the external agent is an XML document that follows the specification described in the SPARQL Query Results XML Format [4]. This file, which contains all the existing solutions for the query, is then parsed and converted in a list. Finally, the back-end will provide each logic solution to the query, one at the time, via its backtracking mechanism.

6 Initial Assessment and Conclusions

Our implementation provides an representation abstraction layer for web ontologies that can be accessed by logic programs. As the Semantic Web is a relatively recent research topic, the technology we used is still undergoing active development and constant change. As expected, the lack of standards at this early stage poses a problem for which no easy solution exists. We did have to make a decision about the adoption of external representation and query languages and this pair ended up being OWL and SPARQL.

More importantly, SPARQL and OWL have been shown to be suitable languages in the scope of our work: to build a working base system which demonstrates how Contextual Logic Programming can be used to represent and query ontologies in a simple yet powerful way.

Future Work As of this writing, the system is still undergoing development, proper benchmarks still need to be deployed, namely comparisons with other ontology representation and reasoning systems as well as SPARQL implementations. Our goal will include the comparison of computations performed on different representation schemes, for instance Prolog internal knowledge base, an external relational database or another SPARQL agent. The initial results are encouraging, as the performance figures appear to be competitive, we shall report on these in another article.

We illustrated how our representation can be used to develop Semantic Web agents by introducing what we called the front-end and the CxLP back-end. However, there are aspects of other systems that may benefit from the ability to query SPARQL sources: we are presently working on integrating the SPARQL back-end into the ISCO [2] system.

This work is the first approach to represent an ontology using CxLP. Currently the formal semantics of OWL DL is not fully represented and should be a focus of research as the study and implementation of the existing reasoning algorithms for OWL DL.

The implemented SPARQL agent currently does not cover the full specification. Although full SPARQL language support is not the intended objective, we are working towards extending it.

Another important goal is to provide the core with capabilities to work with several ontologies at a time. Although it is not relevant for the purpose of this work, it is an essential feature for any Semantic Web application software.

Also, currently, the developed system only allows querying the loaded ontologies. We aim to allow for some form of reasoning over partially known ontologies.

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. http://www.w3.org/TR/rdf-sparql-XMLres/.
5. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic Web. *Scientific American*, 284(5), 2001.
6. DARPA. http://www.daml.org/. DAML+OIL, 3 February 2007.
7. 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/.

8. Leigh Dodds. XML Army Knife. http://xmlarmyknife.org/api/rdf/sparql/query, 5 December 2006.
9. 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.
10. Frank Manola and Eric Miller. Rdf primer. W3C Recommendation, World Wide Web Consortium, February 2004.
11. Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, World Wide Web Consortium, February 2004.
12. Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2006.
13. W3C. SPARQL Protocol For RDF. http://www.w3.org/TR/rdf-sparql-protocol/, 6 October 2006.