Processing OWL2 ontologies using Thea: An application of logic programming

Vangelis Vassilades, Jan Wielemaker, Chris Mungall

Abstract. Traditional object-oriented programming languages can be difficult to use when working with ontologies, leading to the creation of domain-specific languages designed specifically for ontology processing. Prolog, with its logic-based, declarative semantics offers many advantages as a host programming language for querying and processing OWL2 ontologies, and even for building applications. In particular, the SWI-Prolog environment includes an RDF library and has been successfully used to develop full-blown AJAX semantic web applications. However, until now there has been a lack of any library providing direct and complete support for OWL2.

We have developed Thea, a Prolog library that fills that need. Thea uses an RDF library for parsing and serializing ontologies, but the core model is independent of RDF and is based directly on the OWL2 functional-style syntax, allowing direct manipulation of axioms from within Prolog. Thea also offers additional capabilities including SWRL support, a bridge to the java OWL API and translation of ontologies to Description Logic programs.

In this paper we provide examples of using Thea for processing ontologies, and compare the results to alternative methods. Thea is available from GitHub: http://github.com/vangelisv/thea/tree

1 Motivation

The OWL2 language provides a large variety of powerful constructs for building and reasoning over ontologies. These ontologies are typically developed using sophisticated editing environments by domain specialists rather than computer scientists or programmers. However, there is frequently a need to access ontologies or knowledge bases programmatically - in order to perform scripting operations or to build applications. One popular approach is to use RDF toolchains, which provide access at the triple level. There are a variety of such tools for a variety of programming languages. This approach works well for lightly axiomatized linked-data collections, but for working with the TBoxes of heavily axiomatized OWL2 ontologies the triple view can be too low level.

The OWL API[5] is an example of an alternative approach in which the programmer works directly with OWL2 constructs from an axiom-oriented perspective. The API closely follows the OWL specification, making it a natural fit for working with the TBox of complex ontologies. The OWL API is implemented in Java, the language of choice for many enterprise applications. However, there is something of an impedance mismatch between object-oriented (OO) languages and logical axioms (similar to the much-noted impedance mismatch between OO and relational databases). This has motivated the development of domain-specific languages (DSLs)[8] for manipulating ontologies, including the Ontology Pre-Processing Language (OPPL)[2].

However, the creation of a DSL is an onerous task, and it can be difficult to get the balance between expressivity and simplicit correct. An alternative approach is to use an existing high-level declarative language. Ideally this language should be Turing-complete, and should offer pattern-matching and querying capabilities. Here we explore the use of prolog as one such language.

2 Prolog as an Ontology Processing Language

Prolog offers many advantages as a host programming language for working with ontologies, due to it's declarative features and pattern-matching styles of programming[1].

A prolog program is a collection of *horn clauses*, rules of the form **Head:-Body** , where **Head** is a single goal **Body** consists of a number of subgoals joined by conjunctions or disjunctions (written "**,** " or "**;** " respectively) . A clause with an empty body is known as a *fact*. A collection of facts is called a database. Each goal is a predicate combined with zero or more arguments, where the arguments can be variables (which are written using a leading upper-case character), atoms or compound terms. Prolog predicates are typically referenced in **Predicate/Arity** , where Arity is the number of arguments taken by the predicate. Prolog programs make the closed world assumption and implements negation-as-failure.

Prolog goals are typically resolved by chronological backtracking (although other resolution strategies are possible). Prolog has other impure non-logical features such as the *cut* predicate, as well as meta-logical predicates for performing aggregate operations.

Prolog belongs to a family of rule-oriented languages which have been explored as an alternative basis for the semantic web and reasoning, an approach that has been criticised by some in the OWL community[7]. However, here we are more concerned with Prolog as a *programming* language for working with ontologies rather than a direct substrate for ontologies with logic programming semantics.

There are a number of different Prolog implementations. In considering a system for performing programmatic tasks on ontologies certain considerations such as supporting libraries are important. The SWI-Prolog environment[18] has the advantage of providing both RDF/XML parsers and an efficient in-memory triplestore in the form of the **semweb** library[19]. SWI-Prolog has been used to build fully-fledged semantic web applications.

3 Thea: a library for OWL2

3.1 Design Decisions

Our goal was to build a programming library that supported OWL2 directly through the prolog database, rather than indirectly via RDF triples. This was the approach taken by the first version of Thea, developed in 2005 to support OWL as a complement to the SWI-Prolog **semweb** library.

However, this first version took a “frame-oriented” approach, providing a small number of predicates to support the basic entities - classes, properties and individuals. In redesigning Thea to support OWL2 we decided to opt for an “axiom-oriented” approach, and in particular to follow the OWL2 structural syntax[11] specification precisely. Here, every axiom in the ontology would correspond on a one-to-one basis with facts in the prolog database.

3.2 Model

Our model directly corresponds to the OWL2 structural syntax[11] specification, with only minor variations between the two. For example, a simple subclass axiom between two named classes (Human and Mammal) is written using a **subClassOf/2** fact:

```
subClassOf('http://example.org#Human','http://example.org#Mammal').
```

In contrast to many programming languages, there is no need for an extensive API for interrogating these structures, as we can directly query the prolog database using goals with variables as arguments. For example, to find all direct superclasses of Human we would use a variable in the second argument position:

```
?- subClassOf('http://example.org#Human',X).
X = 'http://example.org#Mammal'
```

In the cases where arguments are not named entities, we use prolog terms corresponding to expressions, again with a direct correspondence between the OWL2 specification and prolog functors and arguments. See table1 for a comparison of an axiom stated using both OWL2 structural syntax and in the native prolog form.

Thea2 also allows an optional alternate style called *plsyn*, taking advantage of the ability to define infix operators in Prolog syntax, yielding something similar to Manchester syntax yet native prolog terms (see table1).

Thea also allows for ontology interrogation using strongly-typed predicates such as **subOjectPropertyOf/2** and **subDataPropertyOf/2** . These are implemented as prolog rules.

Thea has support from the Semantic Web Rule Language (SWRL). SWRL antecedent-consequent rules are represented in the prolog database as facts using a two-argument **implies/2** predicate, rather than directly as prolog rules.

OWL2	EquivalentClasses(forebrain_neuron intersectionOf(neuron someValuesFrom(partOf forebrain)))
Prolog	equivalentClasses([forebrain_neuron, intersectionOf([neuron, someValuesFrom(partOf, forebrain)])]).
Plsyn	forebrain_neuron == neuron and partOf some forebrain.

Table 1. Comparison of the representation of an OWL axiom in both OWL2 structural syntax and the native form asserted in the prolog database. Note the minor difference in that where the OWL2 spec allows n-ary predicates to represent sets or lists, we use explicit prolog list syntax (denoted by the square brackets). We also show a more compact prolog representation taking advantage of the ability to declare some predicates as infix in Prolog. Full IRIs are truncated for brevity.

3.3 Concrete Representations: Parsing and Serialization

The OWL2 language has a number of alternative concrete forms, the normative one being RDF/XML, which can be parsed and serialized using the SWI-Prolog semweb library. Thea includes prolog rules for translating between these RDF graphs and the axiom-oriented representation; these rules are based directly on the OWL2 RDF Mapping[3]. There are also parsers and serializers for SWRL and OWL2-XML[10].

In addition it can be very convenient and efficient to read from or write to a native prolog representation, so Thea provides this capability too.

3.4 Reasoning

With Thea it is possible to reason using either Logic Programming techniques, or by bridging to external reasoners.

Description Logic Programs The primary motivation for using Prolog is the declarative programmatic style rather than an alternative fragment of first order logic. However, certain logic programming engines offer useful reasoning capabilities that complement description logic reasoning.

The intersection of logic programming and description logics is known as DLP[4] and used in systems such as KAON2[9]. SWRL rules are also translated directly into prolog . We have implemented this translation as part of Thea, and extended it for certain OWL2 features such as property chain axioms. The resulting logic programs can be evaluated using systems such as XSB, Yap or B-Prolog.

Backward-chaining Many prolog engines use backtracking to evaluate goals. One problem here is that it is possible to write non-terminating programs. Nevertheless there are some circumstances where backtracking can be use safely: for example, if an ontology consists entirely of proper subclass axioms between either named classes or existential restrictions, then backtracking can be a convenient way of interrogating the TBox, even for large ontologies.

Thea provides the ability to do this kind of "scruffy" backward-chaining based reasoning.

External reasoners Thea also includes as an optional component a bridge to the OWL API using the SWI JPL package. This allows seamless access to the extensive capabilities of the OWL API, including access to powerful DL reasoners such as Pellet[16] and FaCT++[17].

Thea also has an interface to DIG servers.

4 Applications of Logic Programming to Ontologies

The use of high level declarative programming languages can be advantageous when working with rich and complex ontology models. Here we present some examples of using prolog plus Thea to perform different tasks.

4.1 Ontology Querying

As noted previously, there is no specific API for interrogating or manipulating OWL2 ontologies using Thea2. The declarative pattern matching and symbol manipulation features of Prolog suffice. In addition it is simple to create new rules, effectively naming queries.

For example, we can define a predicate for determining the least common ancestor (LCA) over the SubClassOf axiom:

```
common_ancestor(X,Y,A) :-
  entailed(subClassOf(X,A)),
  entailed(subClassOf(Y,A)).

least_common_ancestor(X,Y,A) :-
  common_ancestor(X,Y,A),
  \+ ((common_ancestor(X,Y,A2),
       A2\=A,
       entailed(subClassOf(A2,A)))).
```

The **least_common_ancestor/3** predicate can then be re-used in subsequent queries.

Another powerful feature of prolog is the ability to perform meta-logical queries involving aggregation. For example, if we want to summarise all classes by the number of instances asserted to be types of that class we can do this using **aggregate/4** :

```
class(C),aggregate(count,I,classAssertion(C,I),Num).
```

This goal would succeed once for every class **C** , unifying **Num** with the number of individuals in class **C** .

By combining the LCA predicate with aggregate queries it becomes very simple to write *semantic similarity* applications, a popular use of biological ontologies[14]. Thea includes a sample application for calculating these metrics for OWL knowledgebases.

4.2 Ontology Processing

Ontologies are typically created and maintained using development environments such as Protege[13], which provide a graphical user interface to allow domain experts to view, create and edit axioms. In addition to these end-user oriented tools, there is frequently a need to do programmatic processing or scripting of ontologies for tasks that would be tedious and repetitive to do by hand.

Consider a hypothetical ontology that by default follows a strict jointly-exhaustive pairwise-disjoint paradigm, but with occasional exceptions that are explicitly declared using a specified annotation property. We can automate the generation of these axioms using the following goal, which can be evaluated in a failure-driven loop:

```
setof(X,(subClassOf(X,Y),
         \+ annotationAssertion(status,X,unvetted)),
      Xs),
assert_axiom(disjointUnion(Y,Xs))
```

Of course it is possible to write a program to do this in a language such as java using the OWL API, which may be preferable in many circumstances. However, if this a need to perform multiple scripting tasks on ad-hoc basis then a declarative means of processing ontologies can be a useful complementary technique.

The examples directory in the Thea distibution contains many recipes such as this one.

4.3 Label generation

One of the challenges in ontology development is maintaining consistent class labels that conform to community norms[15]. Given the appropriate equivalence axioms it should be possible to auto-generate labels or suggestions for labels.

Prolog Definite Clause Grammars (DCGs) allow for simple configuration of community-specific class labeling rules. For example, given an OWL class expression (here specified using plsyn infix):

```
length and qualityOf some (axon and partOf some pyramidal_neuron)
```

We may like to derive a more user-friendly label such as *length of pyramidal neuron axon*. This label contains less information than the class expression, but is usually sufficient for humans to understand.

We can do this using the following DCG:

```
% non-terminals - class expressions
term(T)                           --> qual_expr(T) ; anat_expr(T).
qual_expr(Q and qualityOf some A) --> qual(Q),[of],anat_expr(A).
anat_expr(P and partOf some W)    --> anat(W),anat_expr(P).
anat_expr(A)                      --> anat(A).

% terminals - named classes
anat(A)                           -->
    {entailed(subClassOf(A,anatomical_entity)),
     labelAnnotation_value(A,Label)},
    [Label].
qual(Q)                           -->
    {entailed(subClassOf(Q,quality)),
     labelAnnotation_value(Q,Label)},
     [Label].
```

We can exploit the non-determinisim of prolog to generate multiple values. For example, if a class has two labels *pyramidal neuron* and *pyramidal cell* then multiple compositional class labels will be generated. This can be very useful for automatically generating labels to be indexed for text search.

The same grammars can be used to parse controlled natural language expressions. This technique has been used for both parsing and label generation in many biological ontologies using Obol grammars[12].

The Thea distribution comes with some examples geared towards biological ontologies.

4.4 Translating to and from other sources

Ontologies can be constructed both manually and automatically. In the latter case, the ontology may be constructed from some other data sources: flat files, XML or relational data.

The pattern matching and rule-driven nature of Prolog make it a good match for data translation tasks.

To take a biological example, given a two-column table mapping types of cell to the markers expressed on the surface of that cell, we can specify the translation to a complex OWL axiom using a single rule:

```
CellType < hasPart some (surface and hasPart some Marker) :-
    cell_marker(CellType,Marker).
```

Many prolog implementations also provide libraries for XML processing and for database connectivity, which means that similar declarative rules such as the above can be specified for these sources too. The Thea distribution includes examples of both.

4.5 Ontology Web Applications and Web Services

In addition to providing an expressive means of querying, processing and performing translations on ontologies, it is possible to write full blown applications using Thea2 via the SWI-Prolog http library. The Thea distribution contains some simple examples, including a basic web-based axiom browser.

5 Comparison with other systems

5.1 SPARQL

The SPARQL language is commonly used for querying ontology-centric linked data, and sometimes for querying the ontology itself (TBox querying). Thus there is some overlap with the querying capabilities of prolog+Thea. However, SPARQL suffers from certain limitations in certain circumstances:

- No means of updating data
- Too RDF-centric for querying complex TBoxes
- Lack of ability to name queries (as in relational views)
- Lack of aggregate queries
- Lack of programmability

There are various extensions to overcome these limitations: SPARUL for updates, SPARQL-DL for OWL-level querying of TBoxes. In addition SPARQL enjoys the distinction of being a W3C standard and is supported by most triplestores. SPARQL engines may also provide efficient query optimization. Nevertheless, sometimes SPARQL does not offer the requisite features to perform certain kinds of queries or translations, such as the ones described in this paper. In these cases the ability to perform queries via Prolog offers a useful complementary tool in the semantic web developers arsenal.

5.2 OPPL

Another means of processing ontologies is using OPPL, a Domain Specific Language designed specifically for this task. The OPPL documentation provides the following example for imposing that all subclasses of gender are disjoint:

```
?x:CLASS, ?y:CLASS
SELECT ?x subClassOf gender,
?y subClassOf gender
WHERE ?x!=?y
BEGIN
ADD ?x disjointWith ?y
END;
```

We can do the same thing using a failure driver loop in Thea:

```
subClassOf(X,gender),
subClassOf(Y,gender),
X\=Y,
assert_axiom(disjointClasses([X,Y]))
```

In this case we can see a close correspondence, with minor syntactic differences. OPPL is perhaps easier to teach, being smaller, and having a familiar SQL-like syntax. OPPL also has the significant practical advantage of currently being integrated with the Protege 4 environment.

However, prolog offers many advantages such as higher expressivity, Turing completeness, the ability to name queries, meta-logical predicates, well-understood semantics etc. Although we are not aware of a full formal specification of OPPL, it appears from the grammar that there are many examples presented in this document that would require some kind of extension to OPPL.

5.3 The OWL API

Without any doubt the most fully featured programmatic interface to OWL2 is the OWL API. Thea offers considerably less capabilities, and in addition the OWL API is a better choice for many software developers, being implemented in Java. However, we believe that for a certain subset of tasks, prolog offers a number of advantages in terms of declarative programming.

One option is to use the OWL API in conjunction with a more declarative JVM language. This is the approach taken by the Lisp Semantic Web (LSW) library[1], which runs on the JVM. In fact Thea also provides a bridge to the OWL API, although this is optional, and the user can work directly with axioms expressed natively in a prolog database.

6 Conclusions

Thea offers support for OWL2 within a Prolog environment. The full structural syntax is supported. Thea can be used to simplify many programmatic tasks associated with ontologies, including ontology querying and processing. In addition, Thea can be used to construct full applications that have dependencies on complex ontologies.

[1] http://svn.mumble.net:8080/svn/lsw/trunk/

Thea is available from GitHub (http://github.com/vangelisv/thea/tree) and from the Thea website (http://www.semanticweb.gr/TheaOWLLib/). At this time use of the full library requires SWI-Prolog, although we hope to soon offer full support for Yap prolog. A subset of features (excluding RDF/XML reading and writing) are available to any ISO-compliant Prolog implementatin

References

1. W.F. Clocksin and C.S Mellish. *Programming in Prolog*. Springer-Verlag, New York, 1981.
2. M. Egana, A. Rector, R. Stevens, and E. Antezana. Applying Ontology Design Patterns in Bio-ontologies. In *proceedings of EKAW*, volume 2008. Springer, 2008.
3. Bernardo Cuenca Grau, Ian Horrocks, Bijan Parsia, Alan Ruttenberg, and Michael Schneider. OWL 2 Web Ontology Language: Mapping to RDF Graphs. http://www.w3.org/TR/owl2-mapping-to-rdf/, 2008.
4. B.N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *Proc. 12th Intl. Conf. on the World Wide Web (WWW-2003)*, 2003.
5. M. Horridge, S. Bechhofer, and O. Noppens. Igniting the OWL 1.1 touch paper: The OWL API. *Proc. OWL-ED*, 258, 2007.
6. M. Horridge, N. Drummond, J. Goodwin, A. Rector, R. Stevens, and H.H. Wang. The manchester owl syntax. *OWL: Experiences and Directions*, pages 10–11, 2006.
7. I. Horrocks, B. Parsia, P. Patel-Schneider, and J. Hendler. Semantic web architecture: Stack or two towers? *Lecture notes in computer science*, 3703:37, 2005.
8. M. Mernik and A.M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.
9. B. Motik and R. Studer. KAON2–A Scalable Reasoning Tool for the Semantic Web. In *Proceedings of the 2nd European Semantic Web Conference (ESWC05), Heraklion, Greece*, 2005.
10. Boris Motik, Bijan Parsia, and Peter F. Patel-Schneider. OWL 2 Web Ontology Language: XML Serialization. http://www.w3.org/TR/owl2-xml-serialization/, 2008.
11. Boris Motik, Peter F. Patel-Schneider, and Ian Horrocks. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. http://www.w3.org/TR/owl2-syntax/, 2008.
12. Christopher J. Mungall. Obol: Integrating language and meaning in bio-ontologies. *Comparative and Functional Genomics*, 5(7):509–520, 2004.
13. N. F. Noy, M. Sintek, S. Decker, M. Crubzy, R. W. Fergerson, and M. A. Musen. Creating semantic web contents with protege-2000. *IEEE INTELLIGENT SYSTEMS*, pages 60–71, 2001.
14. Catia Pesquita, Daniel Faria, Andr O. Falco, Phillip Lord, and Francisco M. Couto. Semantic similarity in biomedical ontologies. *PLoS Comput Biol*, 5(7):e1000443, 07 2009.
15. Daniel Schober, Barry Smith, Suzanna Lewis, Waclaw Kusnierczyk, Jane Lomax, Chris Mungall, Chris Taylor, Philippe Rocca-Serra, and Susanna-Assunta Sansone. Survey-based naming conventions for use in obo foundry ontology development. *BMC Bioinformatics*, 10(1):125, 2009.
16. E. Sirin, B. Parsia, B.C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

17. D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. *Lecture Notes in Computer Science*, 4130:292, 2006.
18. J. Wielemaker. An overview of the SWI-Prolog programming environment. In *13th International Workshop on Logic Programming Environments*, pages 1–16, 2003.
19. J. Wielemaker, G. Schreiber, and B. Wielinga. Prolog-based infrastructure for RDF: scalability and performance. *Lecture notes in computer science*, pages 644–658, 2003.