Deep Integration of Python with Web Ontology Language

Marian Babik, Ladislav Hluchy *

Intelligent and Knowledge-based Technologies Group,
Department of Parallel and Distributed Computing, Institute of Informatics,
Slovak Academy of Sciences
Marian.Babik@saske.sk, Ladislav.Hluchy@savba.sk

Abstract. The Semantic Web is a vision for the future of the Web in which information is given explicit meaning, making it easier for machines to automatically process and integrate information available on the Web. Semantic Web will build on the well known Semantic Web language stack, part of which is the Web Ontology Language (OWL). Python is an interpreted, object-oriented, extensible programming language, which provides an excellent combination of clarity and versatility. The deep integration of both languages is one of the novel approaches for enabling free and interoperable data [1].
In this article we present a metaclass-based implementation of the deep integration ideas. The implementation is an early Python prototype supporting in-line class and properties declaration, instance creation and simple triple-based queries. The implementation is backed up by a well known OWL-DL reasoner Pellet [3]. The integration of the Python and OWL-DL through meta-class programming provides a unique approach, which can be implemented with any metaclass enabled scripting language.

1 Introduction

The deep integration of scripting languages and Semantic Web has introduced an idea of importing the ontologies directly into the programming context so that its classes are usable alongside classes defined normally. This can provide a more natural mapping of OWL-DL than classic APIs, reflecting the set-theoretic semantics of OWL-DL, while preserving the access to the classic Python objects. Such integration also encourages separation of concerns among declarative and procedural and encourages a new wave of programming, where problems can be

* Acknowledgments: The research reported in this paper has been partially financed by the EU within the project IST-2004-511385 K-WfGrid and Slovak national projects, Research and development of a knowledge based system to support workflow management in organizations with administrative processes, APVT-51-024604; Tools for acquisition, organization and maintenance of knowledge in an environment of heterogeneous information resources, SPVV 1025/04; Efficient tools and mechanisms for grid computing (2006-2008) VEGA 2/6103/6.

defined by using description logics [10] and manipulated by dynamic scripting languages [1]. The approach represents a unification, that allows both languages to be conveniently used for different subproblems in the software-engineering environment.

In this article we would like to introduce an early prototype, which implements some of the ideas of the deep integration in Python language [8]. It supports in-line declaration of OWL classes and properties, instance creation and simple triple-based queries [2]. We will emphasize the notion of modeling intensional sets (i-sets) through metaclasses. We will also discuss the possible drawbacks of the approach and the current implementation.

2 Intensional Sets and Metaclasses

Intensional sets as introduced in [1] are sets that are described with OWL DL's construct and according to this description, encompass all fitting instances. A sample intensional set can be defined by using Notation3 (N3) [12] as, e.g. ":Person a owl:Class; rdfs:subClassOf :Mortal". This simply states that Person is also a Mortal. Assuming we introduce two instances, e.g. ":John a :Person and :Jane a :Mortal", the instances of Mortal are both John and Jane. Please note, that N3 is used only for demonstration purposes, the current implementation can also support NTriples and RDF/XML.

Terminology-wise, a metaclass is simply "the class of a class". Any class whose instances are themselves classes, is a metaclass. A metaclass-based implementation of the intensional sets is based on the core metaclass Thing, whose constructor accepts two main attributes, i.e. default namespace and N3 description of the i-set. The instance of the metaclass is then a mapping of the OWL class to the intensional set. Following the above example class Person can be created with a Python construct:

```
Person = Thing('Person', (),
{defined_by: 'a owl:Class; rdfs:subClassOf :Mortal',\
 namespace: 'http://samplens.org/test#'})
```

This creates a Python class representing the intensional set for Person and its namespace. In the background it also updates the knowledge base with the new assertion. The individual John can then be instantiated simply by calling $John = Person()$. This statement calls the default constructor of the class Person, which provides support for asserting new OWL individual into the knowledge base. A similar metaclass is used for the OWL property except that it can not be instantiated. The constructor is used here for different purpose, i.e. to create a relation between classes or individuals. The notion of importing the ontology into the Python's namespace is then a matter of decomposing the ontology into the groups of intensional sets, generating Python classes for these sets and creating the instances.

This kind of programming is also called metaclass programming and can provide a transparent way how to generate new OWL classes and properties. Since

metaclasses act like regular classes it is possible to extend their functionality by inheriting from the base metaclass. It is also simple to hide the complex tasks needed for accessing the knowledge base, reasoner and processing the mappings between OWL-DL's concepts and their respective Python counterparts.

3 Sample session

A sample session shows an in-line declaration of a class *Person*. This is implemented by calling a static method *new* of the metaclass *Thing* (the static method *new* is used to hide the complex call introduced in Sec. 2). An instance is created by calling the *Person's* constructor, which in fact creates an in-line declaration of the OWL individual (*John*). The print statements show the Python's view of the respective objects, i.e. *Person* as a class and *John* as an instance of the class *Person*. We also show a more complex definition of the *PersonWithSingleSon*, which we will later use to demonstrate the reasoning about property assertions.

```
>>> from Thing import Thing
>>> Person = Thing.new('Person',' a owl:Class .')
>>> John = Person()
>>> print Person
<class 'Thing.Person'>
>>> print John
<Thing.Person object at 0xb7d0b50c>
>>> PersonWithSingleSon = Thing.new('PersonWithSingleSon', \
    """ a  owl:Class ; rdfs:subClassOf   [ a  owl:Restriction ;
     owl:cardinality "1"^^<http://www.w3.org/2001/XMLSchema#int> ;
     owl:onProperty :hasSon
                                        ] ;
     rdfs:subClassOf  [ a  owl:Restriction ;
     owl:cardinality "1"^^<http://www.w3.org/2001/XMLSchema#int> ;
     owl:onProperty :hasChild ] .""")
```

A similar way can be used for in-line declarations of OWL properties. Compared to a class declaration the returned Python class can not be instantiated (i.e. returns *None*).

```
>>> from PropertyThing import Property
>>> hasChild = Property.new('hasChild',' a owl:ObjectProperty .')
>>> print hasChild
<class 'PropertyThing.hasChild'>
>>> hasSon = Property.new('hasSon', ' a owl:ObjectProperty ;
rdfs:subPropertyOf :hasChild .')
```

Properties are naturally used to assign relationships between OWL classes or individuals, which can be as simple as calling:

```
>>> Bob = PersonWithSingleSon()
>>> hasChild(Bob, John)
```

Assuming we have declared several instances of the class *Person* we can find them by iterating over the class list. It is also possible to ask any triple like queries (the query shown also demonstrates reasoning about the property assertions).

```
>>> for individual in Person.findInstances():
...     print individual, individual.name
<Thing.Man object at 0xb7d0b64c> Peter
<Thing.Person object at 0xb7d0b50c> John
<Thing.Person object at 0xb7d0b6ec> Jane

>>> for who in hasSon.query(Bob):
...     who.name
'John'
>>> print hasSon.query(Bob, John)
1
```

4 Implementation and Drawbacks

Apart from the metaclass-based implementation of the i-sets, it is necessary to support query answering, i.e. provide an interface to the OWL-DL reasoner. There are several choices for the OWL-DL reasoners including Racer, Pellet and Kaon2 [4, 3, 19]. Although these reasoners provide sufficient support for OWL-DL their integration with Python is not trivial since they are mostly based on Java (except Racer) and although they can work in server-like mode Python's support for standard protocols (like DIG) is missing. Other possibilities are to use Python-based inference engines like CWM, Pychinko or Euler [13–15]. However, due to the performance reasons, lack of documentation or too early prototypes we have decided to use Java-based reasoners. We have managed to successfully use JPype [16], which interfaces Java and Python at native level of virtual machines (using JNI). This enables the possibility to access Java libraries from within CPython. Having the ability to call Java and use all the capabilities of the current version of CPython (e.g. metaclasses) is a big advantage over the other approaches such as Jython or JPE [11, 9]. We have developed a wrapper class, which can call Jena and Pellet APIs and perform the needed reasoning [5, 3]. The wrapper class is implemented as a singleton and interfaces the Python calls to the reasoner and RDF/OWL API and forwards it to the JVM with the help of the JPype.

One of the main drawbacks of the current implementation is the fact, that it doesn't support open world semantics (OWA). Although the reasoner in the current implementation can perform OWA reasoning and thus it is possible to correctly answer queries, the Python's semantics are based on the boolean values. One of the possibilities is to use epistemic operator as suggested in [1], however this is yet to be implemented. Another problem when dealing with ontologies are namespaces. In the current prototype we have added a set of namespaces

that constitute the corresponding OWL class or property description as an attribute of the Python's class. This attribute can then be used to generate the headers for the N3 description. This approach needs further extension to support management of different ontologies. One of the possibilities would be to re-use Python's module namespace by introducing a core ontology class. This ontology class would serve as a default namespace handler as well as a common importing point for the ontology classes.

The other drawback of the approach is the performance of the reasoner, which is due to the nature of the JPype implementation (the conversions between virtual machines imposes rather large performance bottlenecks). This can be solved by extending the support for other Python to Java APIs and possible implementation of specialized client-server protocols.

5 Related Work

To our best knowledge there is currently no Python implementation of the deep integration ideas. There is a Ruby implementation done by Obie Fernandez, however it is not known what is the current status of the project [20].

The most popular existing OWL APIs, that provide programmatic access to the OWL structures are based on Java [5, 7]. Since Java is a frame language its notion of polymorphism is very different than in RDF/OWL. This is usually solved by incorporating design patterns, which make the APIs quite complex and sometimes difficult to use. The dynamic nature of the scripting languages can support OWL/RDF level of polymorphism and thus it is possible to directly expose the OWL structures as Python classes without any API interfaces. One of the interesting Java projects, which tries to automatically map OWL ontologies into Java through Java Beans is based on the ideas shown in [17]. This approach tries to find a way how to directly map the ontologies to the hierarchy of the Java classes and interfaces.

Among the existing Python libraries, which support RDF and OWL, the most interesting in terms of partial integration are Sparta and Tramp, which bind RDF graph nodes to Python objects and RDF arcs to attributes of such objects [21, 6]. The projects however doesn't clearly address the OWL and are mainly considered with RDF. It is thus difficult to evaluate what is the level of support for the inferred OWL models.

6 Conclusion

We have described a metaclass-based prototype implementation of the deep integration ideas. We have discussed the advantages and shortcomings of the current implementation. We would like to note, that this a work in progress, which is constantly changing and this is just a report of the current status. At the time of writing authors are setting up an open source project, which will host the implementation of the ideas presented. There are many other open questions,

that we haven't covered here including integration of query languages (possibility to re-use ideas from native queries [18]); serialization of the ontologies; representation of rules, concrete domains, etc. We hope that having an initial implementation is a good start and that its continuation will contribute to the success of the deep integration of the scripting and Semantic Web.

References

1. Vrandecic, D., Deep Integration of Scripting Languages and Semantic Web Technologies, In Soren Auer, Chris Bizer, Libby Miller, 1st International Workshop on Scripting for the Semantic Web SFSW 2005 , volume 135 of CEUR Workshop Proceedings. CEUR-WS.org, Herakleion, Greece, May 2005. ISSN: 1613-0073
2. Web Ontology Language (OWL), see http://www.w3.org/TR/owl-features/
3. Pellet OWL Reasoner, see http://www.mindswap.org/2003/pellet/index.shtml
4. RacerPro Reasoner, see http://www.racer-systems.com
5. Jena: A Semantic Web Framework for Java, see http://www.hpl.hp.com/semweb/jena2.htm.
6. TRAMP: Makes RDF look like Python data structures http://www.aaronsw.com/2002/tramp, http://www.amk.ca/conceit/rdf-interface.html
7. Bechhofer, S., Lord, P., Volz,R.:Cooking the Semantic Web with the OWL API. 2nd International Semantic Web Conference, ISWC, Sanibel Island, Florida, October 2003
8. G. van Rossum, Computer programming for everybody. Technical report, Corporation for National Research Initiatives, 1999
9. Java-Python Extension, http://sourceforge.net/projects/jpe
10. Baader, F., Calvanese, D., McGuinness, D.,L., Nardi, D. and Patel-Schneider,P., F. editors. The description logic handbook: theory, implementation, and applications. Cambridge University Press, New York, NY, USA, 2003.
11. Jython, Java implementation of the Python, http://www.jython.org/
12. Notation 3, see http://www.w3.org/DesignIssues/Notation3.html
13. Closed World Machine, see http://www.w3.org/2000/10/swap/doc/cwm.html
14. Katz, Y., Clark, K. and Parsia, B., Pychinko: A native python rule engine. In International Python Conference 05, 2005.
15. Euler proof mechanism, see http://www.agfa.com/w3c/euler/
16. JPype, Java to Python integration, see http://jpype.sourceforge.net/
17. Kalyanpur, A., Pastor, D., Battle, S. and Padget, J., Automatic mapping of owl ontologies into java. In Proceedings of Software Engg. - Knowledge Engg. (SEKE) 2004, Banff, Canada, June 2004.
18. Cook, W. R. and Rosenberger, C., Native Queries for Persistent Objects, Dr. Dobb's Journal, February 2006
19. Hustadt, U., Motik, B., Sattler, U.. Reducing SHIQ Description Logic to Disjunctive Datalog Programs. Proc. of the 9th International Conference on Knowledge Representation and Reasoning (KR2004), June 2004, Whistler, Canada, pp. 152-16
20. Fernandez, O., Deep Integration of Ruby with Semantic Web Ontologies, see gigaton.thoughtworks.net/ ofernand1/DeepIntegration.pdf
21. Sparta, Python API for RDF, see http://www.mnot.net/sw/sparta/