Deep Integration of Scripting Languages and Semantic Web Technologies

Denny Vrandečić

Institute AIFB, University of Karlsruhe, Germany
denny@aifb.uni-karlsruhe.de

Abstract. Python reached out to a wide and diverse audience in the last few years. During its evolution it combined a number of different paradigms under its hood: imperative, object-oriented, functional, list-oriented, even aspect-oriented programming paradigms are allowed, but still remain true to the Python way of programming, thus retaining simplicity, readability and fun.
OWL is a knowledge representation language for the definition of ontologies, standardised by the W3C. It reaps upon the power of Description Logics and allows both the definition of concepts and their interrelations as well as the description of instances. Being created as part of the notoriously known Semantic Web language stack, its dynamics and openness lends naturally to the ever evolving Python language.
We will sketch the idea of an integration of OWL and Python, but not by simply suggesting an OWL library, but rather by introducing and motivating the benefits a really deep integration offers, how it can change programming, and make it even more fun.

1 Introduction

One of the main motivations behind the design of Python was to make everyone a programmer [5]. The language tries not to encumber the user with unnecessary choices and strives to keep simple things simple. Python programs are known for their readability and ease of maintenance.

The extension of Python suggested in this paper is not one of yet another library, but rather of how Python – or any other dynamic typed language – can gain from the possibilities offered by the Web Ontology Language OWL [4].

We will lay special emphasize on describing the basic notions of the suggested deep integration of Python and OWL, highlighting two exemplary advantages – the idea of *Intensional Sets* and how *language independence* can be gained from following the suggested approach – but we also take a look at the disadvantages and problems associated with it. In the outlook we will provide a small glance at a possible future, where even more people than today are able to customise and integrate their applications and programs at a much finer grained level.

The basic idea is to let OWL and Python live at the same level. Thus, instead of providing a library able to deal with OWL, we instead suggest how to extend Python naturally, remaining true to the pythonesque idioms but nevertheless gaining all possibilities of Description Logics, actually extending Python by yet another programming paradigm, the logic programming paradigm. We will contrast the standard way of designing an OWL API with the deeply integrated one we suggest.

2 Logic Programming Paradigm

In the 1970s, research in Artificial Intelligence led to the development of the programming language Prolog [2]. Prolog differs fundamentally from then usually used imperative programming languages, because instead of writing the algorithm of how to solve a problem, the problem and its domain are declaratively described and then the run time engine is asked to solve it. This is called the logic programming paradigm. Prolog programs may be considered knowledge bases described in a subset of First Order Logics.

W3C's OWL is based on Description Logics [1], another subset of First Order Logic, widely (but not completely) intersecting logic programs. The point is, that one may declaratively describe the problem domain and data in OWL, using powerful ontologies, created by graphical tools like KAON[1], Protégé[2], WebODE[3] or SWOOP[4] to represent knowledge.

This paper will suggest a deep integration of logics into a standard scripting language, Python, which already is a multi-paradigm language, enabling the user, among others, to use imperative, functional, object- and aspect-oriented techniques. By adding a further paradigm we allow the user to exactly use the tools which fit best to a given part of the problem.

3 Intensional Sets

Instead of *loading* an ontology, the notion of *importing* one leads to a different association for the programmer. The difference may be regarded as pedantic or subtle, but it is an important one: the programmer, instead of regarding the ontology as mere data she has to load and access via an API, the imported ontology behaves like a library, extending her possibilities like only code does.

Let us take a look at the following OWL fragment:

```
<owl:Class rdf:ID="Person">
  <rdfs:subClassOf rdf:resource="&philo;Mortal" />
</owl:Class>
<philo:Person rdf:ID="Plato" />
<philo:Mortal rdf:ID="Socrates" />
```

[1] http://kaon.semanticweb.org
[2] http://protege.stanford.edu/
[3] http://webode.dia.fi.upm.es/WebODEWeb/index.html
[4] http://www.mindswap.org/2004/SWOOP

It states, that `Person` is a subclass of `Mortal`, that `Plato` is a `Person` and `Socrates` a `Mortal`. Importing this OWL fragment into our Python program, we are able to write:

```
from example import *
for anyone in Mortal: process(anyone)
```

Instead of mapping OWL classes to Python classes, we map them to *intensional sets*. Intensional sets are sets that are not defined by individual assertions of membership for all elements, but instead are described with OWL DL's construct and then, according to this description, encompass all fitting instances.

In our example we would not process only `Socrates`, but also `Plato`, who, as a `Person`, is a `Mortal` as well. For simple taxonomic relations like this one the point seems trivial. But with OWL DL one may construct much more complex classes, like the class of all instances that are human and have a mother who is a teacher, or all known instances that are not instances of the class of birds. DL classes may be used as queries over the known data, and the intensional sets allow for a very pythonesque, and thus easy, use of these.

The example could also be solved in a number of more traditional ways:

1. A classic OWL library would allow the loading and querying of ontologies.

```
ontology = load(example)
Mortal = ontology.getConcept("philo:Mortal")
mortals = ontology.getInstancesOf(Mortal)
for anyone in mortals: process(anyone)
```

Mortal is an object that represents the OWL Class `philo:Mortal`, `mortals` is the result set of the query `getInstacesOf(Mortal)` against the ontology.

2. A standard approach to mapping OWL and Python would be mapping OWL classes to Python classes. This would lead to different results:

```
ontology = load(example)
mortals = ontology.getInstancesOf(Mortal)
for anyone in mortals: process(anyone)
```

The loading of the ontology will build Python classes out of the OWL classes described in the ontology (like `Mortal`), thus allowing the use of these classes and easy creation of new instances (`aristotle = new Mortal()`). The query in the second line translates to the same query as in alternative 1, but uses reflection (which is no problem, as classes are objects in Python).

This is not about the amount of code – the difference is much deeper: it is about the different approaches toward the integration of scripting languages and description logics. One can either add the OWL functionality as a library and use it via its interface, or deeply integrate it into the idioms of the scripting language and use the power of description logics transparently.

Intensional sets are a more natural mapping of OWL classes than Python classes or API calls could provide, as OWL's formal semantics is defined on a set theoretical foundation. Adding instances to a set is as easy as adding further individuals to the specified set, making it a proper instance of this OWL class with all the implications that follow.

Naturally one can already achieve all functionality described in this chapter with a careful combination of filters, reflection and merged sets, but instead of using this rather complicated constructions we simply allow the user to choose OWL for the description of his data, enabling the user to take the best tool and language for a specific task: a scripting language for scripting processes and a description logic language for descriptions.

4 Language Independence of Data

The separation of declarative and procedural parts of software into two parts expressed by two different languages, specifically designed to solve their respective tasks, reaps yet another strong benefit: the data of the program is automatically available for different programming languages, tools and systems.

Whereas it is out of scope for this paper to describe the ideas of how to integrate description logics with other languages like Ruby, PHP or Perl, such integration should be possible in an analogue way. This ultimately leads to one of the visions of the Semantic Web becoming reality: the data becomes free and interoperable, and can be used within several frameworks without importers or converters. The programmer is able to set up the program domain declaratively, even using graphical ontology editors, and import them into a program without any further work. Knowledge bases from heterogeneous sources can be integrated with tools like OntoMap or frameworks like FOAM[5]. All this can be done by the non-expert users, provided the tools are easy enough to use. It certainly takes a less technical skilled user than it would take for changing a program.

This is one of the main selling points of the Semantic Web, which would be available to the programmer immediately by using languages enhanced the way described here. Again all this benefits would be also available by mere libraries who offer OWL functionality, but the deep integration leads to a much lower learning effort, as most of the idioms are already used within the language anyway. Just providing an OWL API would lead to a higher exposure of the internal and technical details of OWL and the Semantic Web language stack than necessary.

5 Drawbacks

A number of problems arise from the fundamental choices made for OWL. Those are necessary in the context of the Semantic Web, but do not seem to fit well within the context of a programming language.

The lack of the *Unique Name Assumption* in OWL is not one of them. This means that individuals are not necessarily different because they have different

[5] `http://www.aifb.uni-karlsruhe.de/WBS/meh/foam`

names. Although this idea seems hard to grasp for some modellers when building ontologies, programmers actually are used to calling one instance of an object with more than one variable name anyway. Equality is a relation that is tested explicitly, so this, maybe surprisingly, is no problem at all.

A real problem on the other hand is the *Open World Assumption*. This forces us to assume not to know everything: just because something is not stated does not mean it does not hold. Taking the above mentioned example, we know that `Socrates` is a `Mortal`, but it is not stated explicitly that he is not a `Person` as well.

This may lead to problems:

```
if Socrates in Person: print 'Socrates is a person.'
else: print 'Socrates is not a person.'
```

`Socrates in Person` will evaluate to `False`, thus OWL semantics do not agree with the Python semantics here. The result should not be that `Socrates` is not a `Person`, but that it is not known. This does not map easily on the true/false dichotomy of boolean logics as implemented in programming language semantics.

Probably the most natural mapping would be to implicitly use an epistemic operator when using intensional sets. Thus sets include all *known* instances of a concept. The example would be correctly implemented in the following way:

```
if Socrates in Person: print 'Socrates is a person.'
else: if Socrates in NotPerson: print 'Socrates is not a person.'
      else: print 'I do not know if Socrates is a person.'
```

For this we need another intensional set, `NotPerson`, defined in the OWL file as the complement of Person.

The most obvious disadvantage of combining two languages so deeply is that you have to learn them both. Python aims at a wide audience, being easy to learn for beginners and experts in programming. OWL on the other hand is rather an experts language. RDF/XML-Serialisation [3] did obviously not have easy readability and editability with simple text editors in mind. But there already exist numerous tools for the graphical and easy editing of OWL ontologies, either build on existing knowledge engineering tools, like KAON or Protégé, or build from the ground up for editing OWL ontologies, like SWOOP.

Still, those tools are often aiming at the expert knowledge engineer or do not pass industrial strength quality assurance. Hacking OWL ontologies with a simple text editor or with dedicated XML or even RDF editors can easily lead to hard to debug syntactical errors, thus learning a lot of the fundamentals of the lower levels of the Semantic Web language stack is often still a crucial skill.

We expect that with the effort of the community and the growth of interest in the industrial area new and easy to use ontology editors will start to appear more frequently. The integration described in this paper will immediately benefit from such advancement, allowing programmers to automatically supply editors for large amounts of the data for their programs, without having to write them explicitly.

But description logics, as well as logic in general, do need some training, and if users of different abilities in this field work together on some set of data, understanding the more complicated constructs of description logics may become challenging.

6 Outlook

This paper presented a proposal for a different approach to the combination of a scripting language – Python was chosen exemplary, but the arguments are valid for others as well – with description logics, represented by the Web Ontology Language OWL. As both Python and OWL are powerful and complex languages there are still tons of open questions, that could not be covered here:

- How to construct new intensional sets in Python? Or should we at all?
- Which approach to naming the intensional classes and the instances will be best? How to deal with namespaces? How to manage different ontologies?
- How will changes to the ontology be propagated? How can I save new ontologies with the new entities created programmatically?
- How to represent roles? What about concrete domains?
- What kind of semantics will be taken into account for the equality test of intensional sets? Python semantics default to extensional tests – but maybe intensional equivalence is more appropriate?
- How to combine these ideas with query languages? Would they also be representable as intensional sets?

The biggest task of all is the actual implementation. With this paper we describe our vision of the integration of OWL and Python. We want to start a discussion within the community: do these ideas make sense, how should they be changed, or maybe even abandoned? How to solve the still open issues? Is there a base of interested developers and users? Finally, a stable and strong picture of how the integration will look like will emerge, leading to a new paradigm within the Python language, enabling more users to contribute to projects, and finally, making programming even more fun.

References

1. F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The description logic handbook: theory, implementation, and applications.* Cambridge University Press, New York, NY, USA, 2003.
2. J. Cohen. A view of the origins and development of prolog. *Commun. ACM*, 31(1):26–36, 1988.
3. M. Dean and G. Schreiber. OWL Web Ontology Language Reference, 2004. W3C Recommendation 10 February 2004, available at http://www.w3.org/TR/owl-ref/.
4. M. K. Smith, C. Welty, and D. McGuinness. OWL Web Ontology Language Guide, 2004. W3C Recommendation 10 February 2004, available at http://www.w3.org/TR/owl-guide/.
5. G. van Rossum. Computer programming for everybody. Technical report, Corporation for National Research Initiatives, 1999.