ActiveRDF: object-oriented RDF in Ruby*

Eyal Oren and Renaud Delbru

DERI Galway
firstname.lastname@deri.org

Abstract. Although most developers are object-oriented, programming RDF is triple-oriented. Bridging this gap, by developing a truly object-oriented API that uses domain terminology, is not straightforward, because of the dynamic and semi-structured nature of RDF and the open-world semantics of RDF Schema.
We present ActiveRDF, our object-oriented library for accessing RDF data. ActiveRDF is completely dynamic, offers full manipulation and querying of RDF data, does not rely on a schema and can be used against different data-stores. In addition, the integration with the popular Rails framework enables very easy development of Semantic Web applications.

1 Introduction

The Semantic Web is a web of data that can be processed by machines, enabling them to interpret, combine and use Web data [1, p. 191]. RDF[1] is one of the foundations of the Semantic Web. A statement in RDF is a triple stating that a subject has a property with some value.

Programming in the Semantic Web means programming against RDF data. And although most current developers have an object-oriented attitude, programming in RDF is currently triple-based, and getting from the one to the other is cumbersome.

The development of an object-oriented RDF API has been suggested several times [2,7,8], but developing such an API faces several challenges. Using a statically typed and compiled language like Java does not address the challenges correctly (as explained next). A scripting language such as Ruby on the other hand, allows us to fully address these challenges and develop ActiveRDF[2], our "deeply integrated" [8] object-oriented RDF API.

1.1 Overview

The following example summarises ActiveRDF and its features. The example program connects to a YARS[3] RDF database, creates a person and saves that person into the database:

* This material is based upon works supported by the Science Foundation Ireland under Grants No. SFI/02/CE1/I131 and SFI/04/BR/CS0694.

[1] http://w3.org/RDF/

[2] See http://activerdf.m3pe.org for download and documentation (open source).

[3] http://sw.deri.org/2004/06/yars/

```
1  class Person < IdentifiedResource
2    set_class_uri 'http://xmlns.com/foaf/0.1/Person'
3  end
4
5  NodeFactory.connection(:adapter => :yars, :host => 'm3pe.org')
6
7  renaud = Person.create 'http://activerdf.m3pe.org/renaud'
8  eyal = Person.find_by_firstName 'eyal'
9  renaud.firstName = 'Renaud'
10 renaud.lastName = 'Delbru'
11 renaud.knows = eyal
12 renaud.save
```

The main features of ActiveRDF are the following (Tramp[4] is a comparable dynamic API, but has only the two first features):

1. An RDF manipulation language (domain-specific language) using the terminology from the dataset, e.g. offering `renaud.firstName` in line 9 instead of `Resource.getProperty`.
2. Read and write access to arbitrary RDF data, exposing data as objects, and translating all method invocations on those objects as RDF queries, e.g. `renaud.firstName` and `renaud.save` in line 12.
3. Usage of various data-stores through adapter system, indicated in line 5 with `:adapter => :yars`.
4. Dynamic query methods based on the data properties, for example the `find_by_firstName` in line 8.
5. Data-schema independence: classes, objects, and methods are created on-the-fly from the apparent structure in the data (currently using the URI definition given in `class Person` on line 1–3).
6. Object caching that optimises performance and preserves memory through lazy data fetching.
7. Integration with Rails[5], a popular web application framework for Ruby, putting the Semantic Web on Rails.

1.2 Outline

The rest of the paper proceeds as follows: in Sect. 2 we discuss the challenges in designing an object-oriented RDF API and argue that a dynamic scripting language (as opposed to a static language) is very suitable for such an API. We demonstrate the usage of ActiveRDF in more detail in Sect. 3 and describe the internal architecture in Sect. 4. In Sect. 5 we describe how we used ActiveRDF to create a Web application for browsing arbitrary RDF data with little effort, and we conclude in Sect. 6.

[4] `http://www.aaronsw.com/2002/tramp`
[5] `http://rubyonrails.org`

2 Background

In this section, we introduce scripting languages, describe the challenges that need to be addressed to develop an object-oriented RDF API, and explain why a scripting language such as Ruby is well suited for such an API.

2.1 Scripting languages

There is no exact definition of "scripting languages", but we can generally characterise them as high-level programming languages, less efficient but more flexible than compiled languages [6]:

Interpreted Scripting languages are usually interpreted instead of compiled, allowing quick turnaround development and making applications more flexible through runtime programming.

Dynamic typing Scripting languages are usually weakly typed, without prior restrictions on how a piece of data can be used. Ruby for example has the "duck-typing" mechanism in which object types are determined by their runtime capabilities instead[6] of by their class definition.

Meta-Programming Scripting languages usually do not strongly separate data and code, and allow code to be created, changed, and added during runtime. In Ruby, it is possible to change the behaviour of all objects during runtime and for example add code to a single object (without changing its class).

Reflection Scripting languages are usually suited for flexible integration tasks and are supposed to be used in dynamic environments. Scripting languages usually allow strong reflection (the possibility to easily investigate data and code during runtime) and runtime interrogation of objects instead of relying on their class definitions.

The flexibility of scripting languages (as opposed to statically-typed and compiled languages) allows us to develop a truly object-oriented RDF API.

2.2 Challenges in object-oriented RDF

There are many RDF APIs available in various different programming languages[7], most of them for accessing one specific RDF data-store. Most RDF APIs, such as in Sesame[8], Jena[9] or Redland[10] offer only generic methods such as `getStatement, getResource, getProperty, getObject`.

Using these generic APIs is quite cumbersome, and a more usable API has been advocated several times [2,7,8]. Such an API would map RDF resources to

[6] Ruby also has the strong object-oriented notion of (defined) classes, but the more dynamic notion of duck-typing is preferred.

[7] `http://www.wiwiss.fu-berlin.de/suhl/bizer/toolkits`

[8] `http://openrdf.org`

[9] `http://jena.sourceforge.net`

[10] `http://librdf.org`

programming objects, RDF predicates to methods on those objects, and possibly RDF Schema[11] classes to programming classes. This API would for example not contain `Resource.getProperty` but `Person.getFirstName`.

However, providing such an object-oriented API for RDF data is not straightforward, given the following issues:

Type system The semantics of classes and instances in (description-logic based) RDF Schema and the semantics of (constraint-based) object-oriented type systems differ fundamentally [4].

Semi-structured data RDF data are semi-structured, may appear without any schema information and may be untyped. In object-oriented type systems, all objects must have a type and the type defines their properties.

Inheritance RDF Schema allows instances to inherit from multiple classes (multi-inheritance), but many object-oriented type systems only allow single inheritance.

Flexibility RDF is designed for integration of heterogeneous data with varying structure. Even if RDF schemas (or richer ontologies) are used to describe the data, these schemas may well evolve and should not be expected to be stable. An application that uses RDF data should be flexible and not depend on a static RDF Schema.

Given these issues, we investigate the suitability of scripting languages for RDF data.

2.3 Addressing these challenges with a dynamic language

The development of an object-oriented API has been attempted using a statically-typed language (Java) in RdfReactor[12], Elmo[13] and Jastor[14]. These approaches ignore the flexible and semi-structured nature of RDF data and instead:

1. assume the existence of a schema, because they rely on the RDF Schema to generate corresponding classes,
2. assume the stability of the schema, because they require manual regeneration and recompilation if the schema changes and
3. assume the conformance of RDF data to such a schema, because they do not allow objects with different structure than their class definition.

Unfortunately, these three assumptions are generally wrong, and severely restrict the usage of RDF. A dynamic scripting language on the other hand is very well suited for exposing RDF data and allows us to address the above issues[15]:

[11] `http://www.w3.org/TR/rdf-schema/`
[12] `http://rdfreactor.ontoware.org/`
[13] `http://www.openrdf.org/doc/elmo/users/index.html`
[14] `http://jastor.sourceforge.net/`
[15] We do not claim that compiled languages cannot address these challenges (they are after all Turing complete), but that scripting languages are especially suited and address all these issues very easily.

Type system Scripting languages have a dynamic type system in which objects can have no type or multiple types (although not necessarily at one-time). Types are not defined prior but determined at runtime by the capabilities of an object.

Semi-Structured data Again, the dynamic type system in scripting languages does not require objects to have exactly one type during their lifetime and does not limit object functionality to their defined type. For example, the Ruby "mixin" mechanism allows us to extend or override objects and classes with specific functionality and data at runtime.

Inheritance Most scripting languages only allow single inheritance, but their meta-programming capabilities allow us to either i) override their internal type system, or ii) generate a compatible single-inheritance class hierarchy on-the-fly during runtime.

Flexibility Scripting languages are interpreted and thus do not require compilation. This allows us to generate a *virtual* API on the fly, during runtime. Changes in the data schema do not require regeneration and recompilation of the API, but are immediately accounted for. To use such a flexible virtual API the application needs to employ reflection (or introspection) at runtime to discover the currently available classes and their functionality.

In summary, dynamic scripting languages offer us exactly those properties to offer a virtual and flexible API for RDF data. Our arguments apply equally well to any dynamic language with these capabilities. We have chosen Ruby as a simple yet powerful scripting language, which in addition allows us to leverage the popular Rails framework for easy development of complete Semantic Web applications.

3 Usage

We now show the most salient ActiveRDF features with two examples. Please refer to the manual[16] for more information on the usage of ActiveRDF. Sect. 4 and will explain how the features are implemented.

3.1 Create, read, update and delete

ActiveRDF maps RDF resources to Ruby objects and RDF properties to methods (attributes) on these objects. If a schema is defined, we also map RDF Schema classes to Ruby classes and map predicates to class methods. The default mapping uses the local part of the schema classes to construct the Ruby classes; the defaults can be overridden to prevent naming clashes (e.g. `foaf:name` could be mapped to `FoafName` and `doap:name` to `DoapName`).

If no schema is defined, we inspect the data and map resource predicates to object properties directly, e.g. if only the triple `:eyal :eats "food"` is available,

[16] `http://activerdf.org/manual/`

we create a Ruby object for `eyal` and add the method `eats` to this object (not to its class).

For objects with cardinality larger than one, we automatically construct an array with the constituent values; we do not (yet) support RDF lists and collections.

Creating objects either loads an existing resource or creates a new resource. The following example shows how to load an existing resource, interrogate its capabilities, read and change one of its properties, and save the changes back. The last part shows how to use standard Ruby closure to print the name of each of Renaud's friends.

```
renaud = Person.create('http://activerdf.m3pe.org/renaud')
renaud.methods        ... ['firstName', 'lastName', 'knows', ...]
renaud.firstName      ... 'renaud'
renaud.firstName = 'Renaud'
renaud.save

renaud.knows.each do |friend|
  puts friend.firstName
end
```

3.2 Dynamic finders

ActiveRDF provides dynamic search methods based on the runtime capabilities of objects. We can use these dynamic search methods to find particular RDF data; the search methods are automatically translated into queries on the dataset.

The following example shows how to use `Person.find_by_firstName` and `Person.find_by_knows` to find some resources. Finders are available for all combinations of object predicates, and can either search for exact matches or for keyword matches (if the underlying data-store supports keyword search).

```
eyal = Person.find_by_firstName 'Eyal'
renaud = Person.find_by_knows eyal
all_johns = Person.find_by_keyword_name 'john'
other = Person.find_by_keyword_name_and_age 'jack', '30'
```

4 Architecture

In this section, we give a brief overview of the architecture of ActiveRDF. ActiveRDF follows ActiveRecord pattern [3, p. 160] which abstracts the database, simplifies data access and ensures data consistency, but adjusted for RDF data.

4.1 Overview

ActiveRDF is composed in four layers, shown in Fig. 1:

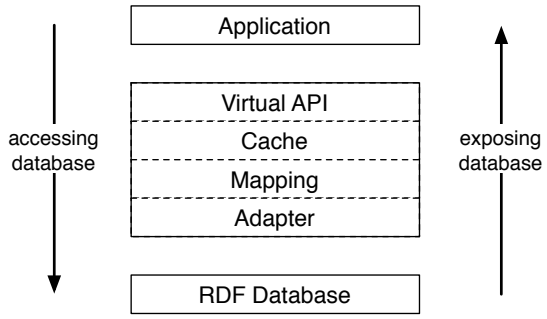

Fig. 1. Overview of the ActiveRDF architecture

Virtual API The virtual API is the application entry point and provides all the ActiveRDF functionality: it provides the domain model with all its manipulation methods and generic search methods. It is not a generated API (hence the name *virtual*) but uses Ruby meta-programming to catch unhandled method calls (such as `renaud.firstName`) and respond to them.

Cache A caching mechanism can be used to reduce access to the database and improve time performance (in exchange for memory).

Mapping Maps RDF data to Ruby objects and data manipulation to Ruby methods. For example, when the application calls a find method or when we create a new person, the mapping layer translates this operation (using an adapter) into a specific query on the data-store. The mapping layer also creates Ruby classes from RDF Schema classes (with methods), or adds Ruby objects (with their own methods) if no schema is available.

Adapter Provides access to a specific RDF data-store by translating generic RDF operations to a store-specific API[17]. The adapter layer enables communication with a specific RDF data-store. Each adapter offers a simple low-level API (consisting of *query, add, remove* and *save*) which is used by the mapping layer. The low-level API is purposely kept simple, so that new adapters can be easily added. The real mapping logic is provided by the generic mapping layer.

4.2 Feature implementation

We now briefly explain how the features mentioned in Sect. 1.1 are implemented in the architecture:

1. The virtual API offers a RDF manipulation language (using Ruby meta-programming) which uses the mapping layer to create classes, objects, and methods which respect the terminology of the RDF data.

[17] in absence of general standardised query language that provides create, read, update, and delete access to RDF data-stores.

2. The virtual API offers read and write access and uses the mapping layer to translate operations into RDF queries.
3. Adapters provide access to various data-stores and new adapters can easily be added since they require only little code.
4. The virtual API offers dynamic search methods (using Ruby reflection) and uses the mapping layer to translate searches into RDF queries.
5. The mapping layer is completely dynamic and maps RDF Schema classes and properties to Ruby classes and methods. In the absence of a schema the mapping layer infers properties of *instances* and adds it to the corresponding objects directly achieving data-schema independence.
6. The caching layer (if enabled) minimises database communication by keeping loaded RDF resources in memory and ensures data consistency by making sure not more than one copy of an RDF resource is created.
7. The implementation design ensures integration with Rails: we have created ActiveRDF to be API-compatible with the Rails framework.

The functionality of ActiveRDF (especially if combined with Rails) allows rapid development of Semantic Web applications that fully respect the principles of RDF.

5 Case Study

We have not yet performed an extensive evaluation of the usability and improved productivity of ActiveRDF (compared to common RDF APIs). Instead we report some anecdotal evidence in our own development of a Web applications that uses ActiveRDF in combination with Rails.

5.1 Semantic Web with Ruby on Rails

Rails is a RAD (rapid application development) framework for web applications. It follows the model-view-controller paradigm. The basic framework of Rails allows programmers to quickly populate this paradigm with their domain: the model is (usually) provided by an existing database, the view consists of HTML pages with embedded Ruby code, and the controller is some simple Ruby code.

Rails assumes that most web applications are built on databases (the web application offers a view on the database and operations on that view) and makes that relation as easy as possible. Using ActiveRecord, Rails uses database tables as models, offering database tuples as instances in the Ruby environment.

ActiveRDF can serve as data layer in Rails. Two data layers currently exist for Rails: ActiveRecord provides a mapping from relational databases to Ruby objects and ActiveLDAP provides a mapping from LDAP resources to Ruby objects. ActiveRDF can serve as an alternative data layer in Rails, allowing rapid development of semantic web applications using the Rails framework.

5.2 Building a faceted RDF browser

We have used ActiveRDF in the development of our prototype faceted browser, available on `http://browserdf.org` and shown in Fig. 2. We have improved faceted browsing, a navigation technique for structured data, with automatic facet construction to enable browsing of semi-structured data [5].

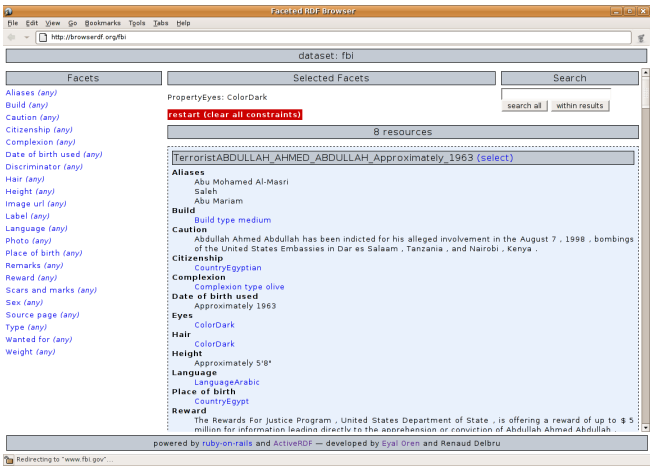

Fig. 2. Faceted browsing prototype

The prototype is implemented in Ruby, using ActiveRDF and Rails. After finishing the theoretical work on the automated facet construction, the application was developed in only several days. In total, there are around 385 lines of code: 250 lines for the controller (consisting mostly of the facet computation algorithm), 35 lines for the data model and 100 lines for the interface which includes all RDF manipulations for display.

6 Conclusion

ActiveRDF is an object-oriented library for RDF data. It can be used with arbitrary data-stores and offers full manipulation of RDF through a virtual API that respects the data terminology. ActiveRDF is completely dynamic and can work without any schema information. Additionally, the integration with the popular Rails framework enables very easy development of Semantic Web applications.

We have analysed the problems of object-oriented access to RDF data and shown that a scripting language is well suited for such a task. In ActiveRDF, we address the challenges as follows:

1. the mapping layer does not rely on the existence of a schema, but can also infer properties from the instance data (as shown in Sect. 1.1),

2. the virtual API is provided dynamically and automatically modified during runtime to stay consistent with a dynamic schema.
3. the mapping layer does not assume the conformance of RDF data to a schema, but instead allows objects with other capabilities than their class definition.

6.1 Discussion

ActiveRDF is by design restricted to direct data manipulation; we do not perform any reasoning, validation, or constraint maintenance. In our opinion, those are tasks for the RDF store, similar to consistency maintenance in databases.

One could argue that statically generated classes have one advantage: they result in readable APIs, that people can program against. In our opinion, that is not a viable philosophy on the Semantic Web. Instead of expecting a static API one should anticipate various data and introspect it at runtime. On the other hand, static APIs allow code-completion, but that could technically be done with virtual APIs as well (using introspection during programming).

6.2 Future work

We have currently released the first version of ActiveRDF and are continuing to improve it. We outline some important issues and how we intend to resolve them. First, multiple inheritance is not yet implemented in the first release, but we are already working on a technique that overrides the internal Ruby type-system, manages our own types and allows multiple inheritance. Secondly, we are removing the cumbersome and technically unnecessary need to pre-define all classes (as shown in the first example in Sect. 1.1). And finally, we plan to do an usage evaluation of ActiveRDF versus other RDF APIs, on (data querying) perfomance and on programmer productivity; we reserve such an evaluation for future work.

References

1. T. Berners-Lee. *Weaving the Web – The Past, Present and Future of the World Wide Web by its Inventor*. Texere, 2000.
2. O. Fernandez. Deep integration of ruby with semantic web ontologies. `http://gigaton.thoughtworks.net/~ofernand1/DeepIntegration.pdf`.
3. M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.
4. A. Kalyanpur, D. Pastor, S. Battle, and J. Padget. Automatic mapping of owl ontologies into java. In *SEKE*. 2004.
5. E. Oren, *et al.* Annotation and navigation in semantic wikis. In *SemWiki in ESWC*. 2006.
6. J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998.
7. D. Schwabe, D. Brauner, D. A. Nunes, and G. Mamede. Hypersd: a semantic desktop as a semantic web application. In *SemDesk in ISWC*. 2005.
8. D. Vrandečić. Deep integration of scripting language and semantic web technologies. In *Scripting for the Semantic Web*. 2005.