A Persistent Programming Language for the Semantic Web

Vadim Eisenberg
Technion
Haifa, Israel
eisenv@cs.technion.ac.il

Yaron Kanza
Technion
Haifa, Israel
kanza@cs.technion.ac.il

ABSTRACT

The *impedance mismatch problem* that occurs when relational data is being used by object-oriented (OO) programs, also occurs when OO programs process RDF data, on the Semantic Web. The impedance mismatch problem is caused by the differences between RDF and the data model of OO languages. In this paper, we present a solution to this problem. Essentially, we modify the OO languages so that RDF resources will become first-class citizens in the OO languages, and objects of OO languages will become first-class citizens in RDF. Three important benefits that follow from this modification are: (1) it becomes natural to use the language as a persistence programming language, (2) the language supports implicit integration of data from multiple data sources, and (3) SPARQL queries and inference can be applied to objects during the run of a program. We present principles of languages that support our solution, and we describe an implementation of our approach, by an extension of the Ruby programming language.

1. INTRODUCTION

The *Semantic Web* is a collection of technologies for managing linked data on the World-Wide Web [6]. In the Semantic Web, data are stored using the RDF data model, where the data items, called *resources*, have *properties* attached to them and can be linked one to another by the properties forming a graph. Thus, applications in Object-Oriented (OO) programming languages over the Semantic Web, need to cope with RDF resources. However, there are inherent differences between objects of OO programming languages and RDF resources. Essentially, differently from objects, RDF resources have URIs, properties and can be members of more than one class. Differently from RDF resources, objects in OO languages have methods and are members of single class. These differences cause an *impedance mismatch problem*, similar to the impedance mismatch between OO languages and relational databases.

In this paper we propose a solution to the impedance mismatch problem. The solution is based on using a unified data model for both RDF and the OO languages. In the unified model, (1) resources can have methods, as in OO programming languages, (2) objects must have URIs and properties, as in RDF, and (3) an object can be a member of more than one class. In a programming language over the unified model, RDF resources become first-class citizens while all the capabilities of object-oriented programming remain.

Since the RDF data model is designed for representing and storing data on the Web, by considering the objects of an application as RDF resources, it becomes natural to store these objects on the Web. Consequently, a programming language that uses the unified model is a *persistent programming language*, i.e. any object can be defined as persistent and can be stored implicitly, without explicit handling of persistence by the programmer.

In addition, using URIs of the objects in the programming language enables creating mapping of the URIs to different data sources. Using this mapping, data integration can be done implicitly, without specifying in the application logic which data source each object belongs to.

The Semantic Web includes tools for querying RDF data (e.g., SPARQL) and for reasoning on the data in RDF - more information can be inferred from the data stored in RDF. Thus, by considering the objects of an application as RDF resources, SPARQL queries and inference can be applied over these objects.

We have implemented an extension to the Ruby programming language, called *Ruby on Semantic Web*, that uses the unified data model. In this paper, we present the principles that guided the design of the language and describe the main implementation details. We present code examples of our extension in the Appendix.

2. BACKGROUND AND FRAMEWORK

In this section, we provide some background. We shortly survey the main technologies of the Semantic Web, illustrate the impedance mismatch problem and present the motivation for persistent programming languages.

2.1 Semantic Web

The vision of the Semantic Web, is to transform the Web into one global database using URIs and common data models. A more modest goal is to use the Semantic Web technologies for Enterprize Information Integration (see [9]) by creating one global database for an enterprize or a closed community, in the form of a *Corporate Semantic Web* [7]. As stated by Agrawal et al. [1]:

> *A significant long-term goal for our community is to transition from managing traditional databases consisting of well-defined schemata for structured business data, to the much more challenging task of managing a rich collection of structured, semi-structured and unstructured data, spread over many repositories in the enterprize and on the Web.*

Using Semantic Web technologies it is possible to map all the data to a common representation, in order to create a

global view on all the data in the organization. Moreover, in the Semantic Web, reasoning tools over ontologies and logic rules enrich the queries on the integrated data.

2.2 Semantic Web Standards

The main data model of the Semantic Web is **RDF** [13], in which all the data is represented by triples *subject-predicate-object*, where both the *subject* and the *predicate* have to be URIs. Using URIs provides a way to uniquely identify resources (objects) in the real world and their properties. The objects in RDF can be linked by the predicates, thus constructing a graph of data.

Additional specifications for the Semantic Web include:

- **RDFS** [12]—a language for specifying classes of the RDF resources, sub-class relationships between classes and constraints on subjects or objects of the predicates.
- **OWL** [5]—a language for specifying ontologies of RDF. In OWL, it is possible to specify assertions about objects, their properties, classes, and relations between them. The semantics of OWL is based on Description Logic.
- **SPARQL** [14] and **SPARQL/Update** [15]—protocols and query/update languages for RDF. The SPARQL "query engines" provide their service by SPARQL endpoints. The endpoints can answer the queries about RDF based either on the RDF model only, or can apply reasoning using RDFS, OWL or rule languages.

2.3 The Impedance Mismatch Problem

The object-relational impedance mismatch problem [17] has been recognized and studied. Evidently, inherent discrepancies exist also between object oriented programming languages and the data model of the Semantic Web [16]. The main discrepancies are:

- **Multiple vs. Single class membership.** In OO, every object is a member of a single class. In the Semantic Web, an *individual* (the object in Semantic Web) may be a member of multiple classes.
- **Dynamic class membership.** In OO, objects cannot change their class. In the Semantic Web, individuals can change their classes dynamically.

The following example illustrates the differences between the OO and Semantic Web models. Consider a person *John*, who is both a student and a chess player. Suppose that we want to write an application which uses the information about *John* and other people. While specifying the fact that an individual is a member of several classes is straightforward in RDF, in OO (for example in Java) a programmer must create a class *StudentChessPlayer* that imlements interfaces *Student* and *ChessPlayer*. In OO, a new class must be created for representing combination of classes. This leads to creating unnatural combinations, e.g. *StudentChessPlayerParentDriver*. This complicates the application by requiring maintainence of multiple classes. In the example of *John*, the problem becomes more evident once *John* goes to a summer internship and becomes a member of a new class - *Employee*. Adding class membership to existing individuals is also straightforward in RDF, however there is no way to state in Java that the class of the object *John* has changed to *StudentChessPlayerEmployee*. As a continuation of our example, consider the case where a software application discovers an additional fact about the world, for example that chess players have rating. Dynamically adding a property *hasRating* to an existing class *ChessPlayer* is straightforward in the Semantic Web, while in Java the definition of the class/interface *ChessPlayer* must be changed offline and the class must be recompiled. (Some programming languages, for example Ruby [8], allow adding methods/fields dynamically.)

2.4 Persistent Programming Languages

In *persistent programming languages* [3], an object of any type can be declared as persistent and its persistency is provided implicitly. Thus, the programmer is freed from explicitly loading or saving data.

The *Orthogonal Persistence Hypothesis* [2] states that: *If application developers are provided with a well-implemented and well supported orthogonally persistent programming platform, then a significant increase in developer productivity will ensue and operational performance will be satisfactory.*

3. PRINCIPLES AND DESIGN FEATURES

In order to solve the impedance mismatch problem between OO and Semantic Web, a programming language for the Semantic Web should have one common data model - a hybrid between the object-oriented model and RDF. In this language, any object should be both a first class citizen of the language and a first class citizen of the Semantic Web.

In particular, the following features should be common to all the objects in the language:

- Every object should be identifiable by a URI, as any resource on the Semantic Web, and should be accessible by a URI.
- There should be a possibility to attach an RDF property to any object.
- Every class, including the RDF classes, should have methods that could be called on the instances of the class. While this feature is the basic one of the OO languages, the RDF resources have no methods. (The *Procedural Attachment* is mentioned as a useful feature in [18]). The methods should have URIs, similar to the properties of RDF and should be serializable in some form in RDF format, similar to the stored procedures in DBMS. The methods and the RDF properties should be accessible in the identical form, using the same language notation.
- Every object could be a member of multiple classes and could change its classes dynamically, as in the Semantic Web. Message dispatch should be done as in the languages with multiple inheritance. Moreover, since the methods will also have URIs, it will be possible to call them explicitly by their URI. It will even be possible to call any method by its URI on any object, regarding of its class. Such behavior could be possible in interpreted languages - if the called method accesses methods and fields (by their names) that are defined in the class, the called method could work OK. Otherwise, a runtime error would occur.

Once every object in the language has URI, the language can be made *orthogonally persistent* [4]. The decision about the object persistence can be made based on the object's

URI. In particular, a decision about where the data representing the object resides should be made based on the URI, for example by using some pattern matching on the object's URI. The decision logic should be separated from the program. This way the programmers could manipulate the data using the URIs only and they could be relieved from the location details of the data. The program will be written as if it works with the whole World Wide Web of data, while the boundaries of this virtual world would be defined by some mapping between URIs and data sources. The data sources and the mapping could be changed without changing anything in the application logic. In addition, the program will be written as if all the data in its virtual world exists in RDF, while in reality the data from different data sources in different data models/formats will be mapped on the fly to RDF.

According to [4], a beneficial design feature of a persistent programming language is *persistence independence* — the code should work in the same way both with persistent and transient data, and should not change if persistence of the data changes. We would add to the definition of [4] that the code should also not depend on where the persistent data comes from/is written to.

The fact that every object and every method has URIs enables running SPARQL queries on all the objects of the language, both the in-memory objects and the ones representing persistent data. This is similar to the LINQ technology [10] that enables running SQL-like queries on the native objects and arrays of a programming language. The fact that SPARQL queries can be run on all the objects in the language makes all the objects "equal citizens" in this aspect and is in agreement with the *persistence independence* property. It is also beneficial that the queries would be language-integrated (first-class citizens of the language), as in the LINQ technology.

Logical inference about the objects must be configurable into the language environment, since it is one of the important features of the Semantic Web. That means that once a Semantic Web reasoner is "plugged" into the language environment, the language operations should be executed considering the inferred information. To illustrate the power of the inference, consider the following example. In RDFS, the *range* is a constraint on the class of the object part of the *subject-predicate-object* property. A property can also be declared as being *sub-property* of another property, for example, *hasBrother* and *hasSister* can be declared as *sub-properties* (IS-A) of a property *hasSibling*. Suppose we have a property *hasSister*, which has a range of *Female* and is a sub-property of a *symmetric* and *transitive* property *hasSibling*. From the following statement of the language: *John.hasSister = Mary*, the language environment can infer that *Mary* is a *Female*, she is a sibling of *John* and *John* is a sibling of *Mary* (due to the fact that *hasSibling* is symmetric). A later query of the *John.hasSibling* property will return *Mary* as a sibling. From the additional statement: *Mary.hasSister = Alice*, the language environment can infer automatically that *Alice* is also a sibling of *John* (since *hasSibling* property is transitive).

4. IMPLEMENTATION

To provide an illustration of programming in a language, designed according to the principles specified in the previous section, we implemented an extension to the Ruby programming language [8], namely *Ruby on Semantic Web*. We decided not to create a new programming language from scratch but to graft the desired features on an existing OO programming language. Our implementation adds to Ruby an approximation of the aforementioned desired features. We built our prototype on top of AcitveRDF[1] — a library for manipulating RDF data. We used implementation ideas from [11]. In addition, we plan to change an existing SPARQL query engine to enable method calling in SPARQL queries, in order to comply with the design demand of adding methods to RDF.

4.1 Ruby Programming Language

The Ruby programming language is a dynamic, interpreted programming language with high meta-programming capabilities. In particular, a Ruby class may be written in several places and the methods can be dynamically added, removed, renamed (via alias mechanism) in any place in the code. The language environment provides "hook methods" for different language events. For example, there are hooks that are called when a new method is added or a missing method is called. The missing method hook, for example, allows the programmer to extend the original method invocation mechanism of the language. In addition, the flexible syntax of Ruby, for example the fact that the parenthesis after the method call are optional, makes Ruby a very powerful language for defining new Domain Specific Languages on top of it.

4.2 Prototype

In our prototype, we had to reconcile between the original objects of Ruby and the Semantic Web resources. The solution we chose is as follows:

- We added URIs to the original objects of Ruby. The URIs for the original objects of Ruby are generated automatically by the language environment. Any object of Ruby and any object of Semantic Web can be accessed by a URI. Any object can have additional URIs assigned to it.
- We added URIs and URI namespaces to Ruby simulating literals of the language.
- We added a "DNS" module that decides where each data item must be loaded from/saved to, according to the URI of the item and according to the preprogrammed rules. The "DNS" module provides separation of the application logic and the logic of where each object should reside.
- We enabled adding Ruby methods to the Semantic Web resources.
- We enabled adding RDF properties to the Ruby objects.

As a result of having RDF properties attached, any object can be a member of multiple Semantic Web classes (in addition to being a member of some native Ruby class). Method dispatch of Ruby was changed (by aliasing all the methods) to consider the methods in all the classes (the native one of Ruby and the Semantic Web classes). In case a method is found in several classes of the object that are not subclasses of one another, an exception is thrown during the runtime.

Additional features that were implemented:

[1] http://www.activerdf.org/

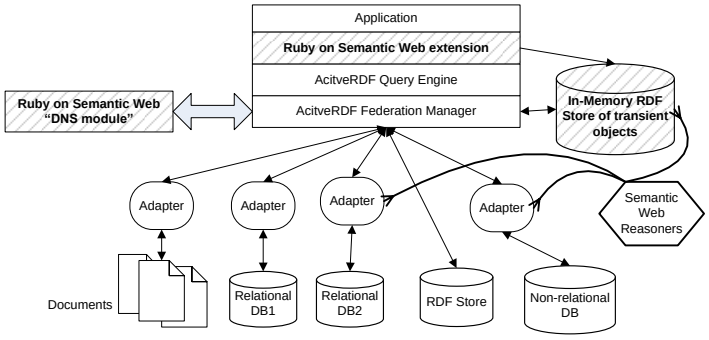

Figure 1: The architecture of Ruby on Semantic Web

- Embedding of SPARQL and SPARQL/Update queries in Ruby, using the original syntax of Ruby
- All the objects in our prototype, both the native Ruby objects and Semantic Web resources, can participate in SPARQL queries, with inference enabled (by plugging in a Semantic Web reasoner).
- The values of the properties can be retrieved and updated via SPARQL or SPARQL/UPDATE, or by applying "." (dot) and "="operators of Ruby, similarly to method calls and assignments.

4.3 Architecture

The architecure of Ruby on Semantic Web is depicted in Figure 1. The Ruby on Semantic Web is built on top of the ActiveRDF architecture [11]. The parts that we added appear hatched.

In the figure, the data in the enterprise reside in several data sources - in documents, in two relational databases, in an RDF Store and in some non-relational database. There are adapters for transforming SPARQL queries into the queries specific for each data source, for example into SQL in case of a relational database. The adapters provide virtual RDF views on all the data in the enterprise. There is no adapter the RDF Store, since it contains the data already in the RDF format.

The adapters are connected to the ActiveRDF Federation Manager - the layer that is responsible for distributing queries among multiple sources and aggregating the results. The arrows are bidirectional meaning that both querying and updating of data sources is enabled. We added a "DNS" module - the module that maps URIs of the objects to data sources in order to determine which data source the new data should be written to. An additional data source — an in-memory RDF store is used for representing information about the objects during the runtime, in RDF. This data source is used for running queries over the original objects of Ruby (the queries about the resources of the Semantic Web run on the regular data sources of the enterprise).

The Semantic Web reasoners can be plugged to the in-memory RDF store and to the adapters of other data sources in order to enable logical inference over the data. The Ruby on Semantic Web uses ActiveRDF Query Engine engine for running queries. The Ruby on the Semantic Web adds information about every new object to the in-memory RDF store in order to enable running SPARQL queries on the Ruby objects. Once an object is garbage-collected, our extension removes the information about the collected object from the store.

The application that runs on top of Ruby on Semantic Web is unaware of all the layers beneath the Ruby on Semantic Web. The application works with URIs and is unaware about where the data resides, what are the formats of the data and which data items are stored and which are inferred by the reasoners. The application can access any object by URI and can run SPARQL queries on all the objects.

5. FUTURE WORK

Future work includes developing optimization techniques for the proposed programming language, such as caching, materialized views etc. Another question could be how to add to the language advanced persistence features such as persistence of the program state and of local variables.

6. REFERENCES

[1] R. Agrawal et al. The Claremont report on database research. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 37(3):9–19, Sept. 2008.

[2] M. Atkinson. Persistence and java - a balancing act. In *Objects and Databases*, volume 1944 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 2001.

[3] M. Atkinson et al. The Object-Oriented Database System manifesto. In *Proceedings of the 1st Intl. Conf. on Deductive and Object-Oriented Databases*. Kyoto, Japan, Dec. 1989.

[4] M. P. Atkinson et al. An orthogonally persistent java. *SIGMOD Rec.*, 25(4):68–75, 1996.

[5] S. Bechhofer et al. OWL Web Ontology Language reference. Recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[6] T. Berners-Lee, J. A. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[7] R. Dieng-Kuntz. Corporate Semantic Webs. *Encyclopaedia of Knowledge Management, D. Schwartz ed, Idea Publishing Group*, 2005.

[8] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly, Cambridge, 2008.

[9] A. Y. Halevy et al. Enterprise Information Integration: successes, challenges and controversies. In *SIGMOD '05*, pages 778–787, New York, NY, USA, 2005.

[10] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In S. Vansummeren, editor, *PODS 2006*, page 706, Chicago, Illinois, June 2006.

[11] E. Oren et al. ActiveRDF: Object-Oriented Semantic Web Programming. In *WWW '07*, pages 817–824, New York, NY, USA, 2007.

[12] RDF Core Working Group. RDF vocabulary description language 1.0: RDF schema. Recommendation, W3C, Feb. 2004. http://www.w3.org/TR/rdf-schema/.

[13] RDF Core Working Group. Resource description framework (RDF): Concepts and abstract syntax. Recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[14] RDF Data Access Working Group. SPARQL query language for RDF, Jan. 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115.

[15] A. Seaborne et al. SPARQL Update. A language for updating RDF graphs. Member submission, W3C, July 2008. http://www.w3.org/Submission/SPARQL-Update/.

[16] Semantic Web Best Practices and Deployment Working Group. A Semantic Web primer for Object-Oriented software developers. Working group note, W3C, Mar. 2006. http://www.w3.org/TR/sw-oosd-primer/.

[17] J. D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice-Hall, 2008.

[18] Web Ontology Working Group. OWL Web Ontology Language Use Cases and Requirements. Technical report, W3C, Feb. 2004. http://www.w3.org/TR/webont-req/.

APPENDIX

Code Examples

1. Defining a namespace *family* for the URI prefix `'http://www.example.org/family#'`

   ```
   namespace 'http://www.example.org/family#', :family
   ```

2. Retrieving/updating properties:
   The following code retrieves the values of `family::hasSalary` property of resources `family::John` and `family::Jane` and updates the property `family::commonIncome` of the resource `family::DoeFamily`. The namespace *family* is registered to designate the prefix
   `http://www.example.org/family#`, as in XML. The double colon is used according to the syntax of Ruby, in contrast to the namespace syntax of XML and some Semantic Web representations.

   ```
   namespace 'http://www.example.org/family#', :family
   family::Doe.family::commonIncome = family::John.family::hasSalary + family::Jane.family::hasSalary
   ```

   After the code is executed, a triple (family::Doe, family::commonIncome, <the calculated encome>) is added to the data source, according to the rules in the "DNS"' module regarding the *family* namespace.
   For comparison, the following code of ActiveRDF [11] accomplishes the same task:

   ```
   Namespace.register :family, 'http://www.example.org/family#'
   john = RDFS::Resource.new 'http://www.example.com/family#John'
   jane = RDFS::Resource.new 'http://www.example.com/family#Jane'
   doe = RDFS::Resource.new 'http://www.example.com/family#DoeFamily'
   doe.family::commonIncome = john.family::hasSalary + jane.family::hasSalary
   ```

   In the code of ActiveRDF, the objects representing the Semantic Web resources must be explicitly created before accessing their properties.

3. Using values of properties of RDF subjects in operations of Ruby:

   ```
   commonIncome = family::John.family::hasSalary + family::Jane.family::hasSalary
   theGroomAndBrideParents = family::John.family::hasParent + family::Jane.family::hasParent
   ```

   The values of the properties are considered as scalar for a *functional* property (a property that can have only one value - the example above assumes that a person has only one salary). Otherwise, the values of the properties are considered arrays (as the *hasParent* property in the example above).
   In the example above the integer addition is performed on scalar values of *hasSalary* property of *family::John* and *family::Jane* and their common income (an integer) is calculated. As opposed to handling a functional property, the expression *family::John.family::hasParent* designates an array of the values of the property *family::hasParent*. The Ruby addition of arrays operation is applied to the arrays of the values, producing an array containing the objects representing the parents of *John* and *Mary*.

4. Setting a property *family::hasSister* of *family::John* to be *family::Mary*

   ```
   family::John.family::hasSister = family::Mary
   ```

   this statement deletes all triples with subject *family::John* and predicate *family::hasSister* from the data target specified by the "DNS", according to the URI of the subject or the predicate. After deleting the previous values, a triple *(family::John, family::hasSister, family::Mary)* is added to the same data target.

5. Adding new values to the property *family::hasSister* of *family::John*

   ```
   family::John.family::hasSister +=[ family::Mary, family::Alice]
   ```

   here the Ruby's addition of arrays is used (the property *family::hasSister* is not functional , so it can have multiple values - the array semantics is applied)

6. Using values of properties of RDF subjects in the Ruby control flow:

   ```
   example::company.example::hasEmployee.each { |employee|
     if(employee.family::hasSalary < 1000) then
          employee.family::hasSalary += 100
     end
   }
   ```

   The code above iterates over all the employees of *example::company* and increases the salary of each employee by 100, if the employee's salary is less than 1000. Here a Ruby iterating construct (*each* method with a block) is used.

7. Running SPARQL queries embedded in Ruby:
   Consider the following query in SPARQL (we omitted the definition of *dbpo*, *dbpp* and *dbpr* namespaces):

   ```
   select ?film ?budget ?actor
   where {
      ?film dbpo:starring  dbpr:Bruce_Willis;
            dbpo:budget ?budget;
            dbpo:starring ?actor .
       ?actor dbpp:birthPlace dbpr:United_Kingdom
   }
   ```

This query returns all the films, which has as stars Bruce Willis and some other actor who was born in the United Kingdom. The films, their budgets and the actors born in the UK are returned by the query.
The query can be programmed in our extension of Ruby in the following way:

```
results =
  select q::film q::budget q::actor where {
    q::film dbpo::starring  dbpr::Bruce_Willis;
            dbpo::budget q::budget;
            dbpo::starring q::actor .
    q::actor dbpp::birthPlace dbpr::United_Kingdom
  }
```

In the Ruby version a prefix "q::" is used instead of the "?" sign of SPARQL, in order to comply with the parsing rules of Ruby.
For comparison, the following code of ActiveRDF [11] accomplishes the same task:

```
results = Query.new.select(:film,:budget,:actor).
                  where(:film,  DBPO::starring, DBPR::Bruce_Willis).
                  where(:film, DBPO::budget,:budget).
                  where(:film, DBPO::starring,:actor).
                  where(:actor, DBPP::birthPlace, DBPR::United_Kingdom).execute
```

In the code of ActiveRDF, the query is created by calling *select*, *where* and *execute* methods of a Query class, while in our code the query is written almost AS-IS, without the need to explicitly call any methods.

8. The SPARQL queries can be further integrated in other Ruby constructs, for example in iterations:

```
totalBudget = (select q::film where {
       q::film dbpo::starring  dbpr::Bruce_Willis;
       dbpo::starring q::actor .
       q::actor dbpp::birthPlace dbpr::United_Kingdom
}) .inject (0) { |sum, film| sum += film.dbpp::budget }
```

Here the *inject* method of Ruby is used for iteration over all the films in which Bruce Willis and some actor born in the UK starred, and for calculation of their total budget

9. Reasoning over the properties :

```
family::john.family::hasSister = family::mary
print family::john.family::hasSibling # prints family::mary since hasSister is a
                                      # subproperty of hasSibling
print family::mary.family::hasSibling # prints family::john, since hasSibling is a symmetric property
print family::mary.rdf::type # prints Female as one of the types, since the range of hasSister is Female
family::mary.family::hasSister = family::alice
print family::john.family::hasSibling # now both family::mary and family::alice are printed as siblings
                                      # of family::john, since hasSiblig is a transitive property
```

10. The inference of the types and changing types dynamically:

```
family::John.family::hasSalary = 100
print family::john.rdf::type # assuming the domain of the property hasSalary is Employee,
                             # one of the  printed types of family::john is Employee
family::John.family::studiesAt = [technion::cs]
print family::john.rdf::type # assuming the domain of the property studiesAt is Student, both Employee
                             #  and Student types are printed
family::John.family::hasSalary = nil # all the values of the property hasSalary are removed
print family::john.rdf::type # now family::john is not Employee anymore, since he has no value for
                             # hasSalary property (assuming there are no other evidence that he is
                             # an Employee), he is only a Student
family::John.family::studiesAt = [] # all the values of the property studiesAt are removed
print family::john.rdf::type        # now family::john ceases to be a Student,
                                    # since he has no value forstudiesAt property (assuming
                                    # there are no other evidence that he is a Student)
```

11. Defining persistent methods for persistent classes and demonstrating the message dispatch:

```
class example::Student
   def printTitle
       print "I am a Student"
  end
end
```

Here a persistent method *printTitle* is added to the persistent class *example::Student*. The method is serialized in RDF in some data target, according to the "DNS". Both the message "printTitle" and the method *printTitle* in the class *family::Student* are represented in RDF and have URIs assigned to them. The method can be called either by its message

name as usual - “printTitle”, or by the URI of the message or by the URI of the method. In the case of a call by the URI of the method the dispatch is made statically - the specified method is called. Consider the following hierarchy - suppose there are classes Person, Employee, Student and Manager. Employee and Student are sub-classes of Person and Manager is a sub-class of Employee. Suppose that each class has a “printTitle” method that prints “I am “ + the class name string. The following code will produce the output as described (considering the type inference as in the previous example):

```
# suppose the initial type of family::john is Person
family::john.printTitle # "I am Person"

family::john.family::hasSalary = 100 # now family::john is Employee
family::john.printTitle # "I am Employee"

family::john.example::isManagerOf = [family::bob] # now the type of family::john is Manager, assuming
                                                   # the domain of example::isManagerOf
                                                   # property is Manager
family::john.printTitle # "I am Manager"

family::john.family::studiesAt = [technion::cs]
family::john.printTitle # an exception is thrown - two candidate methods exists -
                        # one in the Manager and
                        # another one in the Student class

# calling the specific method by its URI, "I am a student" is printed
family::john.URI("'http://www.example.org/family#Student.Methods.printTitle')
family::john.printTitle # "I am Student"

family::john.family::studiesAt = []  # family::john ceases to be Student
family::john.example::isManagerOf = [] # family::john ceases to be Manager
family::john.printTitle # "I am Employee"

family::john.family::hasSalary = nil  # family::john ceases to be Employee
family::john.printTitle # "I am Person"
```

12. Running SPARQL queries on the original Ruby objects:

```
class ChessPlayer # an original Ruby class
...
end

# the following query returns all the Ruby objects of type ChessPlayer
players = select q::player where {
        q::player rosw::rubyType ChessPlayer
      }
```

13. Linking between native Ruby objects and Semantic Web individuals, and querying about them:

```
player = ChessPlayer.new # create a native object of Ruby
family::john.family::hasSister += [player] # the native object player is linked to
                                           # the individual family::john via
                                           # the family::hasSister property
# The following query returns all the sisters of family::john that are also ChessPlayers
sisterChessPlayers = select q::sister where {
        family::john family::hasSister q::sister .
        q::sister rosw::rubyType ChessPlayer
      }
```

14. Assigning a URI to a native object of Ruby

```
player = ChessPlayer.new
...
example::abc = player # assigning a URI example::abc to the object pointed by player
player.play() # calling the play() method of the player
example::abc.play() # calling the same method of the same object,
                    # using its new URI
...
# another run of another program
example::abc.play() # calling the play() method of
                    # the previously serialized object player
```

```
# the object is implicitly deserialized
```

After the assignment above, the object, pointed by the variable *player* will have an additional URI - *example::abc*. The object will be serialized in a data target according to the rules defined in the "DNS" module for the objects of the *example* namespace. The methods of the object could be called later (during the current run of the program) or even in the later runs by using the new URI. During the later runs of the program, the object will be trasparently deserialized and its methods could be called as usual. The example shows implicit serialization/deserialization