Programming Applications
over the Semantic Web

Vadim Eisenberg
Programming Applications over the Semantic Web

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Vadim Eisenberg

Submitted to the Senate of the Technion — Israel Institute of Technology
Elul 5772 Haifa September 2012
The research thesis was done under the supervision of Dr. Yaron Kanza in the Computer Science Department.

Part of this work was published in proceedings of:

- the 27th International Conference on Data Engineering (ICDE), 2011, pages 1324-1327: Vadim Eisenberg and Yaron Kanza. Ruby on Semantic Web

I would like to express my gratitude to my advisor, Dr. Yaron Kanza for his excellent guidance during my studies. I am thankful to Yaron for providing me clear directions and focus in research, for his constant encouragement and support, and for freedom he gave me in choosing to develop my research work in my own ways, while insisting upon high standards.

I would like to thank my examiners—Prof. Shmuel Katz and Prof. Avigdor Gal, for their constructive and insightful comments regarding my work, which helped me to improve the final version of this thesis.

I would also like to thank my former colleagues from IBM, and fellow graduate students in the Technion—Roy Levin and Alex Kogan, for their support and for having interesting discussions and spending some nice time during the studies.

The generous financial support of the Technion is gratefully acknowledged.
# Contents

Abstract .................................................. 1
Abbreviations and Notations ................................ 2

1 Introduction ............................................. 3

2 Background .............................................. 7
   2.1 The Semantic Web .................................... 7
       2.1.1 Syntactic Interoperability ...................... 10
       2.1.2 Semantic Interoperability ...................... 12
       2.1.3 Corporate Semantic Web ...................... 15
   2.2 Federated Information Systems ...................... 15
   2.3 Impedance Mismatch .................................. 20
   2.4 Persistent Programming Languages ................. 22
   2.5 The Ruby Programming Language .................. 23

3 The SWOOM model ....................................... 24
   3.1 The OO-RDF Impedance Mismatch .................... 24
   3.2 Principles of SWOOM ................................. 26
       3.2.1 URIs as Language Literals ..................... 26
       3.2.2 URIs and Access Transparency ................. 28
       3.2.3 Location, Migration and Replication Transparency .. 28
       3.2.4 Property Attachment ......................... 29
       3.2.5 Procedural Attachment ......................... 30
       3.2.6 Multiple Dynamic Class Membership .......... 31
       3.2.7 Orthogonal Persistence ......................... 32
       3.2.8 Model Transparency ......................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.9</td>
<td>Persistence Independence</td>
<td>34</td>
</tr>
<tr>
<td>3.2.10</td>
<td>Language-Integrated Queries</td>
<td>35</td>
</tr>
<tr>
<td>3.2.11</td>
<td>Integrated Logical Inference</td>
<td>36</td>
</tr>
<tr>
<td>3.2.12</td>
<td>Transaction Management</td>
<td>38</td>
</tr>
<tr>
<td>4</td>
<td>Ruby on Semantic Web</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>The Architecture</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>The Implemented Features</td>
<td>43</td>
</tr>
<tr>
<td>4.2.1</td>
<td>URIs as Language Literals</td>
<td>43</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Persistence Independence</td>
<td>47</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Language-Integrated Queries</td>
<td>52</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Location and Migration Transparency</td>
<td>56</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Integrated Logical Inference</td>
<td>58</td>
</tr>
<tr>
<td>4.2.6</td>
<td>URIs and Access Transparency</td>
<td>60</td>
</tr>
<tr>
<td>4.2.7</td>
<td>Property Attachment</td>
<td>61</td>
</tr>
<tr>
<td>4.2.8</td>
<td>Orthogonal Persistence</td>
<td>69</td>
</tr>
<tr>
<td>4.2.9</td>
<td>Procedure Attachment</td>
<td>70</td>
</tr>
<tr>
<td>4.2.10</td>
<td>Multiple Dynamic Class Membership</td>
<td>76</td>
</tr>
<tr>
<td>4.2.11</td>
<td>Transaction Management</td>
<td>85</td>
</tr>
<tr>
<td>4.2.12</td>
<td>Model Transparency</td>
<td>86</td>
</tr>
<tr>
<td>5</td>
<td>Optimizations</td>
<td>87</td>
</tr>
<tr>
<td>5.1</td>
<td>Aggregation Pushing</td>
<td>87</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Implementation</td>
<td>89</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Experimental Evaluation</td>
<td>90</td>
</tr>
<tr>
<td>5.2</td>
<td>Condition Pushing</td>
<td>92</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Experimental Evaluation</td>
<td>93</td>
</tr>
<tr>
<td>5.3</td>
<td>Join Pushing</td>
<td>94</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Experimental Evaluation</td>
<td>96</td>
</tr>
<tr>
<td>5.4</td>
<td>Bulk Update</td>
<td>97</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Experimental Evaluation</td>
<td>98</td>
</tr>
<tr>
<td>5.5</td>
<td>Query Prefetching and Results Caching</td>
<td>99</td>
</tr>
<tr>
<td>5.6</td>
<td>Value Propagation for Query Prefetching</td>
<td>99</td>
</tr>
<tr>
<td>5.7</td>
<td>Redundant Query Elimination by Reasoning</td>
<td>100</td>
</tr>
<tr>
<td>5.8</td>
<td>Value Propagation by Reasoning</td>
<td>101</td>
</tr>
<tr>
<td>5.9</td>
<td>Summary</td>
<td>102</td>
</tr>
</tbody>
</table>
6 On the Effectiveness of Programming in Ruby-S

6.1 Accessing a Property ........................................... 104
6.2 Updating a Property .............................................. 105
6.3 Performing a SPARQL query .................................... 107
6.4 Reasoning and Property Attachment .......................... 109
6.5 Discussion ......................................................... 113

7 Conclusion .......................................................... 115

Abstract in Hebrew .................................................. i
List of Figures

4.1 The architecture of Ruby-S . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

5.1 The impact of pushing aggregation into SPARQL and SQL. The labels on the X axis specify where the main computation was performed, Ruby-S or SPARQL. The words in parentheses specify where the aggregation was performed, Ruby-S, SPARQL or SQL. The Y axis presents the running time, in centiseconds, on a logarithmic scale. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
List of Tables

2.1 OO artifacts vs. Relational artifacts . . . . . . . . . . . . 20
## Listings

3.1 URIs as literals in a programming language .......................... 27  
4.1 Namespaces in Ruby-S ................................................. 44  
4.2 Accessing properties by the dot operator ........................... 47  
4.3 The SPARQL query performed by the execution environment for Line 4 of Listing 4.2 .............................................. 48  
4.4 Properties and assignment ........................................... 50  
4.5 Properties and arithmetic operators ................................. 51  
4.6 A SPARQL query embedded in Ruby-S ............................... 52  
4.7 The SPARQL query in the original SPARQL syntax, that corresponds to the query embedded in Ruby-S in Listing 4.6 53  
4.8 A SPARQL query with a variable, embedded in Ruby-S ... 53  
4.9 A SPARQL query integrated with a loop ......................... 54  
4.10 A SPARQL query with multiple variables and multiple triples in the triple pattern ........................................... 54  
4.11 SPARQL query embedded in the language and the equivalent version with parentheses and receivers specified ........... 55  
4.12 “DNS” specification .................................................. 57  
4.13 Example of accessing an inferred property ....................... 59  
4.14 Property attachment to Ruby objects ............................. 61  
4.15 Assigning a URI to a Ruby object ................................ 65  
4.16 A procedure added to a class of RDF individuals .............. 70  
4.17 Calling a method from a SPARQL query ........................ 74  
4.18 Example of method dispatch in Java .............................. 77  
4.19 Multiple class membership and method dispatch .............. 81  
4.20 An ontology for Listing 4.19 ........................................ 83  
5.1 Summation on the results of a query ............................... 88  
5.2 Optimized SPARQL query for the code shown in Listing 5.1 88
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>Optimized SQL query for the code shown in Listing 5.1</td>
<td>89</td>
</tr>
<tr>
<td>5.4</td>
<td>Condition inside an iteration over query results</td>
<td>92</td>
</tr>
<tr>
<td>5.5</td>
<td>Optimized version of the code shown in Listing 5.4</td>
<td>93</td>
</tr>
<tr>
<td>5.6</td>
<td>The simulated manually Conditional Push optimization</td>
<td>94</td>
</tr>
<tr>
<td>5.7</td>
<td>“Procedural” join on the results of two SPARQL queries</td>
<td>94</td>
</tr>
<tr>
<td>5.8</td>
<td>Optimized version of the code shown in Listing 5.7</td>
<td>95</td>
</tr>
<tr>
<td>5.9</td>
<td>The manually simulated Join Push optimization</td>
<td>96</td>
</tr>
<tr>
<td>5.10</td>
<td>Update of a property inside a loop</td>
<td>97</td>
</tr>
<tr>
<td>5.11</td>
<td>Optimized version of the code shown in Listing 5.10</td>
<td>97</td>
</tr>
<tr>
<td>5.12</td>
<td>SPARQL/UPDATE version of the code shown in Listing 5.11</td>
<td>98</td>
</tr>
<tr>
<td>5.13</td>
<td>The simulated manually Bulk Update optimization</td>
<td>98</td>
</tr>
<tr>
<td>5.14</td>
<td>Using a variable with known values inside a SPARQL query</td>
<td>100</td>
</tr>
<tr>
<td>5.15</td>
<td>Redundant SPARQL query</td>
<td>101</td>
</tr>
<tr>
<td>5.16</td>
<td>Value Propagation by Reasoning</td>
<td>101</td>
</tr>
<tr>
<td>6.1</td>
<td>Accessing a property using the SPARQL Client library</td>
<td>105</td>
</tr>
<tr>
<td>6.2</td>
<td>Accessing a property in ActiveRDF</td>
<td>105</td>
</tr>
<tr>
<td>6.3</td>
<td>Accessing a property in Ruby-S</td>
<td>105</td>
</tr>
<tr>
<td>6.4</td>
<td>A SPARQL/Update statement.</td>
<td>106</td>
</tr>
<tr>
<td>6.5</td>
<td>Performing the SPARQL/Update statement of Listing 6.4 in plain Ruby</td>
<td>106</td>
</tr>
<tr>
<td>6.6</td>
<td>Performing the SPARQL/Update statement of Listing 6.4 in Ruby-S</td>
<td>106</td>
</tr>
<tr>
<td>6.7</td>
<td>A SPARQL query that finds RDF individuals whose name is “John”</td>
<td>107</td>
</tr>
<tr>
<td>6.8</td>
<td>Implementing the SPARQL query using the SPARQL Client library</td>
<td>107</td>
</tr>
<tr>
<td>6.9</td>
<td>Implementing the SPARQL query using ActiveRDF</td>
<td>107</td>
</tr>
<tr>
<td>6.10</td>
<td>Implementing the SPARQL query in Ruby-S</td>
<td>107</td>
</tr>
<tr>
<td>6.11</td>
<td>A SPARQL query to find Dr. John</td>
<td>108</td>
</tr>
<tr>
<td>6.12</td>
<td>Implementing the SPARQL query using the SPARQL Client library</td>
<td>108</td>
</tr>
<tr>
<td>6.13</td>
<td>Implementing the SPARQL query in ActiveRDF</td>
<td>108</td>
</tr>
<tr>
<td>6.14</td>
<td>Implementing the SPARQL query in Ruby-S</td>
<td>109</td>
</tr>
<tr>
<td>6.15</td>
<td>Calculating siblings in Ruby</td>
<td>110</td>
</tr>
<tr>
<td>6.16</td>
<td>Ontology that defines $\text{hasSibling}$, $\text{hasBrother}$ and $\text{hasSister}$ properties</td>
<td>112</td>
</tr>
</tbody>
</table>
6.17 Setting brother and sister relations and calculating siblings in Ruby-S
Abstract

One of the hardest problems in programming applications for information processing, is the *impedance mismatch*—the inherent discrepancy between data models and models of popular programming languages, for example between the relational model and the Object Oriented model. The Semantic Web technologies—RDF, RDFS, OWL, SPARQL—can be beneficial for information processing, in particular for information integration. However, the impedance mismatch, which stems from inherent differences between RDF and OO, still remains. In this work we illustrate a solution to the impedance mismatch problem—a hybrid model between RDF and OO. Essentially, we modify an OO language so that RDF data items become first-class citizens of the language, and objects of the language become first-class citizens of RDF. The benefits of the hybrid model and of the modified programming language are: (1) it becomes natural to use the language as a persistent programming language, relieving the programmers from handling persistence issues explicitly, (2) tools from both models, such as optimizers, syntax checkers, query and reasoning engines, can be applied to the artifacts of the unified model, (3) programmers in the modified language will have to understand only one conceptual model instead of two, and will be able to mix, without translation, artifacts from both models. In this work we present the hybrid model and the modified programming language, explain their benefits, evaluate code examples written in the programming language, provide examples of possible code optimizations and illustrate the effectiveness of these optimizations.
## Abbreviations and Notations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS</td>
<td>DataBase Management System</td>
</tr>
<tr>
<td>DIMHRS</td>
<td>Defense Integrated Military Human Resources System</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>DOD</td>
<td>Department of Defense (of the United States of America)</td>
</tr>
<tr>
<td>EHR</td>
<td>Electronic Health Record</td>
</tr>
<tr>
<td>ETL</td>
<td>Extract, Transform and Load</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>LINQ</td>
<td>Language Integrated Query</td>
</tr>
<tr>
<td>OLAP</td>
<td>On-Line Analytical Processing</td>
</tr>
<tr>
<td>OO</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>OWL</td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational-DataBase Management System</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Definition Framework</td>
</tr>
<tr>
<td>RDFS</td>
<td>RDF Schema</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>SPARQL</td>
<td>SPARQL Protocol and RDF Query Language</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SWOOOM</td>
<td>Semantic-Web Object Oriented Model</td>
</tr>
<tr>
<td>URI</td>
<td>Universal Resource Identifier</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XPATH</td>
<td>XML PATH Language</td>
</tr>
<tr>
<td>XQUERY</td>
<td>XML Query Language</td>
</tr>
<tr>
<td>WWW</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>W3C</td>
<td>The World Wide Web Consortium</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Effective processing of information is one of the most important tasks in Software Engineering. Effective programming of information processing applications is difficult due to the problem known as impedance mismatch. Impedance mismatch refers to the existence of inherent discrepancy between a model of a programming language (such as Object Oriented) and a data model (such as Relational).

The Semantic Web technologies facilitate information processing, in particular information integration of structured and unstructured heterogeneous data residing in multiple data sources, on the Web. They do so by using URIs, graph based data model (RDF), ontologies and reasoning tools, for providing a common model of the integrated data and for mapping data from different sources to the common model. However, the use of the Semantic Web causes an impedance mismatch between the Object Oriented programming paradigm and the data model of the Semantic Web.

This work aims at finding a solution to the impedance mismatch between the Object Oriented (OO) paradigm and RDF. We propose to replace two models—OO and RDF, by one hybrid model that unifies OO and Semantic Web. We call this model SWOOM. In SWOOM, objects of an OO programming language are first-class citizens of the Semantic Web and the data items of the Semantic Web are first-class citizens of the programming language. As a result, both the programming language tools (syntax checkers, refactoring tools, optimizers) and the Semantic Web tools (reasoners, query processors) can be applied to objects and to data items. Additional beneficial characteristic of SWOOM, is that a programming language in SWOOM is a persistent
programming language. In a persistent programming language, persistence is managed implicitly and the programmer is relieved from the burden of explicitly managing the tasks of loading and storing data items.

We created an example implementation of SWOOM—a programming language that we call “Ruby on Semantic Web”, abbreviated Ruby-S. We developed Ruby-S as an extension of the Ruby programming language by using meta-programming capabilities of Ruby.

Our programming language can be used for writing applications for processing federated data that reside in multiple data sources. It provides the following beneficial features:

- **Persistence orthogonality**—the programmer is relieved from explicitly handling persistence.
- **Persistence independence**—the same code can be used for both persistent and transient artifacts, improving code reuse.
- **Access transparency**—all the data are accessed in a uniform way.
- **Model transparency**—all the data can be queried using one query language—SPARQL, embedded in our programming language.
- **Location, migration and replication transparency**—the application is agnostic to where the data reside, whether they are migrated and whether they are replicated in several data sources.

For comparison, the traditional approach for processing RDF in OO applications is by applying an RDF-to-OO mapping that exchanges the RDF individuals to objects, e.g., ActiveRDF [49]. In such systems, the RDF individuals are mapped to “proxy” objects that represent them during runtime, without handling the impedance mismatch problem. Our approach has the following advantages over a mapping.

First, treating RDF individuals as first-class citizens of a programming language allows compilers, and other language tools, to exploit special features of RDF for increasing programming productivity and for improving performance, e.g., using these features for compiler optimizations, syntax checks and refactoring. Adding methods to RDF enriches the RDF framework, in the same way that stored procedures enrich relational databases in an RDBMS.
Second, by using SWOOM, the language becomes a persistent programming language. This is because RDF and related Semantic-Web technologies are designed for storing data on the Web, by leveraging existing Web protocols such as HTTP and SOAP. Thus, by considering the objects of an application as RDF individuals, it becomes natural to store them on the Web, or on a limited version of the Web, within corporate boundaries.

Furthermore, adding URIs to the objects of a programming language enables associating between objects and data sources. Such associations facilitate data integration by representing in a natural way the mapping of objects to data sources.

A third advantage of our approach is the ability to directly apply in the language Semantic-Web tools for querying RDF and for reasoning over the data.

Due to the above reasons, our solution is not another mapping between RDF and OO. It is a unified model—a hybrid of RDF and OO—where artifacts from both models are first class citizens of the language. In the proposed language, reasoning and querying can be applied over both the objects of the language and RDF individuals. Furthermore, the same code that is used for processing in-memory objects can also be used for processing RDF individuals on the Web. The model also provides a synergy between language tools, e.g., SPARQL queries and OWL ontologies can include references to methods, and methods can embed SPARQL queries and apply reasoning, implicitly.

In order to demonstrate benefits of the fact that Semantic Web artifacts are first class citizens of the programming language, we propose optimizations that utilize properties of the unified model. Our experiments illustrate performance improvements that can be gained by such optimization techniques.

To illustrate the effectiveness of programming over SWOOM, we performed code evaluation of small programming tasks implemented using Ruby-S, ActiveRDF, a library for SPARQL query processing, and also plain Ruby. We show that the implementations in Ruby-S can be done using shorter and simpler code than in all other implementations, for all the examples.

In the following chapters we present the related background for our work, introduce the principles of the proposed hybrid OO-RDF model (SWOOM), describe the Ruby-S programming language and its execution environment,
propose and evaluate optimizations for SWOOM, provide code evaluation of Ruby-S, and, finally, conclude and discuss future work.
Chapter 2

Background

In this chapter we provide background for our work. We describe Semantic Web technologies, Federated Information Systems, Impedance Mismatch, Persistent Programming Languages and the Ruby programming language. We present these topics because the programming language we developed is a Persistent Programming Language, based on the Ruby programming language. It is intended to resolve the Impedance Mismatch problem between the Semantic Web and the Object Oriented programming model. The language is intended to be executed in a runtime environment which is a kind of a Federated Information System.

2.1 The Semantic Web

In this chapter we describe the Semantic Web technologies and the problems the Semantic Web intends to solve. We present two kinds of interoperability problems that exist in processing heterogeneous data sources and describe how Semantic Web technologies solve these interoperability problems. We also present the Corporate Semantic Web as a sub-area of the Semantic Web. The programming language we developed is intended for developing applications that process data over multiple heterogeneous data sources in a Corporate Semantic Web.

The Semantic Web [16] represents a new approach to publishing and processing distributed data and knowledge, by using common formats, existing Web standards and applying ideas from the technologies of the Web,
of databases and of knowledge management. The data and the knowledge can be distributed either on the World Wide Web (WWW), on the Internet, or in a “private Web” within enterprise or community boundaries, behind the firewall.

The Semantic Web abstraction represents a hybrid between a Web, a distributed database and a distributed knowledge management system. An alternative name for the Semantic Web is a Web of Data\(^1\) [12]. As a distributed database, the Semantic Web can be queried by a query language. As a Web, humans can browse the data and programs can crawl, index and search it. In the same way that documents on the Web contain links to other documents, data items on the Semantic Web contain links to other data items. Human users can browse data items on the Semantic Web either by standard Web browsers or by special Semantic Web browsers. The users are able to follow links between data items in the same way they follow links on the ordinary World-Wide Web. As a knowledge base, the Semantic Web represents knowledge about the data (the semantics of the data—hence the name Semantic Web). Knowledge is explicitly captured in ontologies which are published on the Semantic Web, beside the data. Ontologies specify classes of data items, properties of members of classes and relationships between classes. Using ontological reasoning, new facts can be inferred based on existing data. Data can be validated according to ontologies in order to detect errors. An important feature of the Semantic Web is that data items can link to the ontologies describing the data, so the ontologies can be browsed, crawled, indexed, searched and queried inseparably from the data. Thus, a better alternative name for the Semantic Web could be a Web of Data and Knowledge, as mentioned by Good and Wilkinson [36].

The proposed implementation by the World Wide Web Consortium of the Semantic Web vision is based on existing Web standards such as URIs and HTTP and on the existing Web architecture. The idea is to “weave” the Web of Data and Knowledge into the current World-Wide Web of HTML documents. The Semantic Web standards are described shortly in the following sections.

An additional important feature of the Semantic Web is allowing the Web of Data and Knowledge to grow in an evolutionary and decentralized way,

\(^1\)This name however, neglects the knowledge management features that are part of the Semantic Web vision.
without any a priori agreement regarding common data schemata. Every data publisher on the Semantic Web can use existing published vocabularies for describing her data, however she is also free to invent her own terms for her data items. The vocabulary terms of different publishers can be linked one to another, specifying that some term is *the-same-as*, or is *a-kind-of*, another term. This way a *Web of terms*, or a *Web of vocabularies* can be created in addition to the Web of data. Such Web of terms can facilitate query processing on the Semantic Web by allowing software applications programmed using one set of terms to process data that was formulated using some other set of terms. Consider a scenario in which a software application discovers data items on the Semantic Web described in terms that are unknown to the application (the application was not programmed using these terms). The application can search the Semantic Web for the term descriptions or can follow links from the data items to the term descriptions. The application can try to find dynamically the mapping between the unknown terms and the terms it was programmed to use.

The following example shows how an application can process data it was not originally programmed to process. Suppose a programmer writes an application that performs a query $Q$ in terms of some schema $A$, and she runs the application on the data specified by some other schema $B$. When the mainstream database technologies are used, the query must exactly match the schema. So in our example, the query will return no results, since the query and the schema do not match. In contrast to the mainstream database technologies, in the Semantic Web scenario, the application can try to find a schema mapping between $A$ and $B$ on the Web. Once such mapping is found, the application can automatically reformulate\(^2\) $Q$ into a query in terms of schema $B$, and to apply the reformulated query to the data. Actually, the application does not have to find a schema mapping directly from $A$ to $B$. It would suffice to find a set of mappings $S_1 \leftrightarrow S_2$, $S_2 \leftrightarrow S_3$, ..., $S_{n-1} \leftrightarrow S_n$, such that $S_1 = A$ and $S_n = B$. The application can reformulate $Q$ into a query in terms of schema $B$ by sequentially reformulating the query from $A$ to $S_2$, from $S_2$ to $S_3$, ..., and from $S_{n-1}$ to $B$. An alternative to reformulating

\(^2\) *Query reformulation* refers to the exchange of a query in one query language to an equivalent query in a different language, as opposed to *query rewriting* where a query is transformed into an equivalent query in the same language, frequently for utilizing existing views, for optimization purposes.
the query is transforming the queried data from schema \( B \) to schema \( A \) and applying query \( Q \) to the transformed data.

2.1.1 Syntactic Interoperability

In this subsection we describe the first kind of interoperability issues that Semantic Web technologies resolve. The Semantic Web enables software applications to search, crawl and mine several heterogeneous data sources concurrently. To that end, the applications must be able to access and parse all the data in all the data sources, without necessarily understanding the meaning of the data. This is called *syntactic interoperability*.

Syntactic interoperability can be achieved by transforming all the data to some common data format—the *lingua franca* for data. Once all the data is transformed to the common data format, they can be queried and processed using a single language. The transformation can be done either on-the-fly or off-line. Either the data itself can be transformed, or the transformation is performed “virtually”, by reformulating queries from the common format to the formats of the data sources.

In principle, SQL and the Relational data model [58] could provide the common format. Selecting the common model, however, should also be based on how convenient would it be to work with this model on a large scale. The relational data model and SQL are not well suited to represent data on the Web due to name collisions, e.g. it is probable that many different data sources will have tables with common names such as FLIGHTS or ITEMS_IN_STOCK. It would be hard to differentiate between tables from different schemata provided by different data sources. In addition, the relational data model is best suited for processing structured data ordered in tables. Processing unstructured distributed data via relational views could be inconvenient. The relational model is less convenient for information integration than graph based models such as RDF [44]. Even when several data sources use the same column names for their tables, they can have different tables design—the columns can be distributed differently between the tables. Merging and linking between graphs of data items can be done in a natural way. On the contrary, merging data items distributed between tables of different relational data sources, is a more complex task. Additional drawbacks of the relational model for data integration are a lack of
a standard for serialization and exchange, and limitations in representing incomplete data or missing attributes in the integrated data sources.

XML and XML-based standards such as XQUERY [20] and XPATH [26] overcome the names collision problem by using namespaces [21] for elements and attributes. This way, element and attribute names form URIs and every element/attribute in the world can be uniquely identified. However, the data items and the values of attributes in XML still can be regular strings and can introduce name collisions. XML is designed for representing semi-structured, distributed pieces of data. The data model of XML is a tree and is convenient for representing hierarchical data.

The RDF data representation [44] is the basis of the Semantic Web. It represents data in terms of simple facts or simple sentences—subject-predicate-object or, alternatively called, subject-property-value triples. A triple represents the smallest piece of information. The Semantic Web is a collection of triples distributed between different data sources.

RDF improves XML. First, it requires every property to be a URI. In XML using namespaces for tags is optional. A more important requirement is to use URIs for data items (the subjects) themselves. This way every data item in the world can be uniquely identified. In addition, URIs can be HTTP URLs and can be used as a uniform mechanism for accessing and browsing data items (subjects or objects). Once the URI is looked up by a machine or a human, some representation of the data item can be returned by the data publisher, using either a machine-processable or a human readable representation.

Values in RDF are either URIs or literals. When the values are URIs, they can appear in other triples as subjects. So subjects can be pairwise linked by properties, forming a graph. In this graph the subjects and values form vertices, and properties form edges between a subject and another subject, and between a subject and a value. This way a Web of data can be created, which humans can browse and machines can crawl, following the links between data items. Giant Global Graph [14] (GGG) is another name, coined for the Semantic Web. The graph model is generic in the sense that it can represent both relational, hierarchical and unstructured data. In addition, merging and linking between RDF graphs can be done in a natural way, especially due to the fact that the vertices that represent subjects have globally unique identifiers—URIs.
SPARQL [51] is a query language for RDF. SPARQL queries are formulated in terms of triple patterns, using a SELECT ... FROM ... WHERE structure which resembles SQL. In addition to being a query language, SPARQL is also a protocol for querying data sources on the Web using standard Web protocols such as HTTP [32] and SOAP [48]. The data sources that can be queried by the SPARQL protocol are called SPARQL endpoints.

To summarize, RDF is a natural data model for representing and integrating heterogeneous data, distributed between multiple sources on the Web. By using RDF as the common data model, syntactic interoperability between heterogeneous data sources and applications processing them can be achieved and the utility of enabling keyword search on all the data can be provided. In addition to syntactic interoperability, the main power of the Semantic Web comes from the ability to provide semantic interoperability, as described in the next section.

2.1.2 Semantic Interoperability

With syntactic interoperability data published on the Web by one program (the publisher) can be parsed, crawled, searched and visualized by another program (the consumer), even when both programs were developed by different programmers without any a priori agreement between them.

However, the challenge for the consuming program is to process the data in a more meaningful way, for example to pose queries more complex than simple keyword based queries. For such processing, the vocabulary and the meaning of the terms used by the publishing program must be “known” to the consuming program. That is, the meaning of the terms (the semantics) must be programmed beforehand in the consuming program’s code. In other words, either the publishing and consuming programs must be designed together, or they both must use some common standard vocabulary and apply the same meaning to the used terms. Since explicitly designing all the programs on the Web is not feasible, standards could provide semantic interoperability, but they currently fail to do so.

There are various standard XML schemata defined for various domains, such as ebXML [2] for electronic business or HL7 V3 [3] for healthcare informatics. Ideally, standard schemata for all kinds of data could have been created so that all the programs in the world would be developed
according to these standard schemata, achieving the vision of the Semantic Web and the ultimate interoperability between all the programs in the world. However, this ideal is far from being achieved in the real world since there is a lack of one commonly agreed standard in many areas of information systems. Different competing *de jure* and *de facto*, *open* and *proprietary* standards [59] are promoted by different organizations. These organizations pursue various contradicting interests and represent different points of view, due to cultural, technological and other differences. West [59] explains the role of standards in information systems. West [60] describes the economic realities of open standards in information technology. An example of multiple standards is seen in the area of Electronic Health Records (EHR). Eichelberg et al. [27] describes three different standards and an *integration profile* developed for EHR. Due to the current situation with the standards, software applications are not developed according to one common standard.

A realistic solution to the semantic interoperability could be to allow the programs to be developed independently and to use different vocabularies, while providing effective means for mapping between the vocabularies. The differences in vocabularies can be minimized by using standards. However, the differences can still exist due to multiple standards or due to different versions of the same standard.

Transformation of data from one vocabulary to another can be done either (1) programmatically, by implementing components usually called *adapters* or *wrappers*, or (2) declaratively, by letting the programs that consume the data perform translation according to some declared mapping. The declarative way is to state that a term in vocabulary $X$ is equivalent to some other term in vocabulary $Y$, or is a kind of some term in vocabulary $Y$. *RDFS* [22]—RDF Schema, allows to state that some property is a sub-property of another property and some class is a sub-class of another class. This way a declarative mapping between different RDF properties can be provided.

A program that discovers some RDF data specified in terms of a vocabulary that the program cannot understand, can try to find a mapping to the vocabularies it can understand. Here “understanding” of a vocabulary by the program simply means that the semantics of the vocabulary was coded into the program. The program can find the mapping either on the Web by searching or crawling, or in some repository of mappings. These mappings
can be specified either in a document provided by the data provider or can
be specified in a separate document provided by a third party. The program
can even try to find a transitive mapping via several mappings between
different vocabularies.

RDF and RDF Schema provide a way to create a Web of linked data
and of linked schemata, while the data and the schemata can also be linked
together. The programs can traverse this Web of data and process the data
in a meaningful way. This can be done even when the data is specified in
terms of vocabularies that are not a priori known to the programs.

The next step in meaningful processing of data on the Web is to enable
programs to validate the published data or to infer new data from the pub-
lished data. To achieve this, the semantics of the data can be published
in addition to the data itself. The semantics of the data or the domain
knowledge can be represented by ontological axioms and logical rules. Col-
lections of ontological axioms and rules are called ontologies. OWL [11] is a
specification for stating ontological axioms on top of RDFS. SWRL [39] is
a specification for stating logical rules, on top of OWL. Since the ontologies
are built on top of RDF and RDFS, they use URIs, and thus can form a
Web of ontologies, or a Web of knowledge, in addition to the Web of data.
Special components for reasoning over the ontologies, called reasoners, were
developed. Programs that process data can use reasoners to provide more
meaningful data processing, e.g. for validating the data or for inferring new
facts about the data. SPARQL query engines can enrich their answers using
ontologies, by providing inferred results in addition to the results based on
the original data only. Using rules, complex mappings between vocabularies
of data and complex domain knowledge facts can be specified.

To summarize, using Semantic Web technologies a Web of data and a
Web of knowledge can be created, with data and knowledge (the seman-
tics) linked together. These “Webs” can be browsed by humans, and can
be automatically processed by computers in a meaningful way—crawled,
searched, queried, reasoned about and validated. Computer programs can
process data in a meaningful way, even when they were not a priori pro-
grammed to process the data. The data, the vocabularies and the programs
consuming published data can evolve independently, without the producers
and consumers of the data having to a priori agree between themselves on
the format of the data.
2.1.3 Corporate Semantic Web

The Semantic Web vision can be achieved first when limited to the bound-
daries of one organization—a company, an institution or a closed community. An organization can provide incentives for its members to publish data, it can enforce using RDF and organization-wide ontologies. The scalability issues associated with the Semantic Web can be somewhat alleviated since for a single organization the processing is done on a limited scale and not on the whole Web. Issues of privacy and trust are of lesser concern inside an organization, within firewall boundaries. Pashke et al. [50] and Wood [7] described the issues related to Corporate Semantic Webs [50].

2.2 Federated Information Systems

The programming language we developed, Ruby-S, is intended to be exe-
cuted as a part of a Federated Information System [23], in order to process data that reside in multiple heterogeneous data sources. In this section we explain what are Federated Information Systems, why they are needed and compare them with a Data Warehouse [40], which is an alternative approach for processing data that reside in multiple heterogeneous data sources.

The multiplicity of the data sources in enterprises stems from mergers between companies and from acquisitions of some companies by other companies. An additional cause to the multiplicity is specialization of data sources. Different applications are developed or purchased by organizations for handling different aspects of their business. These different applications often use different data management solutions, so sets of disconnected data sources are created for different applications. In the past the benefits of having integrated data and applications were probably underestimated, so enterprises often did not invest in creating enterprise-wide infrastructures for data and applications. It seems the enterprises gave their departments free hand in developing and purchasing applications suiting their needs.

A famous example of multiple applications is personnel and payroll sys-
tems of the US Department of Defense (DOD). Different services of the department (Army, Navy, Marines, Air Force, and also National Guard, Army Reserve, etc.) use different applications for different aspects related to personnel and payment. All together the DOD uses more than 90 systems.
An additional problem is the heterogeneity of the data sources. The information in enterprises resides in databases, spreadsheets, text documents, XML and proprietary file formats. The information is either structured, semi-structured or unstructured. For databases, various relational, pre-relational (hierarchical, network) and post-relational—NoSQL [56] databases are used.

As stated in the Claremont report on Database Research [5]—a report produced by leading figures in the database research community:

“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 enterprise and on the Web.”

An ideal solution for information integration would be to transform all the data in the enterprise into one data source under one schema, to get rid of multiple data sources and heterogeneity. However, an important reality of the enterprise software systems is that once a system (applications and database) are in place and working, in many cases they will continue to exist and will seldom be rewritten or replaced, due to low cost-effectiveness of the rewrite. In most cases, the companies will prefer to maintain existing systems, even if the systems were implemented using obsolete technologies. For databases, it means that once a database is operational and there are multiple applications working with the database—the database itself and its schema, in most cases, are never changed. Ullman and Widom [58], p.486, describe this legacy-database problem:

“... once a database has been in existence for a while, it becomes impossible to disentangle it from the applications that grow up around it, so the database can never be decommissioned.”

So the reality in enterprise information systems is that they have to cope with multiple heterogeneous data sources. The enterprises, however, often have to process the data in several data sources simultaneously, for example posing a query that joins data from several data sources or to
perform data mining on several data sources. Processing the data in multiple heterogeneous data sources could be a hard task for programmers—they have to write an application for such a task in terms of multiple protocols, multiple query languages and multiple schemata. The previously mentioned US Senate hearing refers to a task force of DOD that concluded in 1996 that multiple systems for personnel and payroll in DOD

“...cause significant functional shortcomings... and excessive development and maintenance costs.”

The recommendation of the task force was that

“...DOD should move to a single, all-service, all-component, fully integrated personnel and pay system with common core software and maintenance costs.”

It seems that the enterprises would benefit from integrating all the data source into one source with one format and one schema, while leaving the original data sources and legacy applications working with them in place.

There exist two approaches for data integration: (1) data warehouse and ETL (2) federated information system.

In the data warehouse approach, the data is extracted periodically from all the data sources, transformed into one format and one schema and loaded into one data source. This process is called ETL—Extract, Transform, Load. Updates of data in the warehouse can be made between the loading activities. Applications over the data will process the transformed copies. The data in the warehouse can be organized in a way that is optimized for typical tasks performed by the applications. For example, aggregating data, using OLAP techniques [25]. An additional feature of data warehouses is that they can process multiple versions of data, including both up-to-date and historical data.

In federated information systems, the data are not copied or transformed. A virtual view on all the data sources is created. A federated database provides an abstraction of all the data residing in one data source, under one schema and in one format. The queries posed over the federated database are reformulated to comply with the format and the schemata of the federated sources. Some query processing is done in the sources and some is done by the federated database. Differently from a data warehouse, the data sources
can be updated via a federated database. (A data warehouse contains read-only transformed copies of the original data.)

An important feature of federated databases is that the data can be processed simultaneously by applications working on the data sources and by applications working on the federated database, while preserving *Global Serializability* of the federated data sources. Global Serializability means that the united schedule of all the schedules in all the federated data sources is serializable. Raz [52] describes an algorithm for achieving global serializability.

While federated information systems can transfer some data, in advance, from data sources into a federation site (caching data) for optimization purposes, no transfer of all the data is required. In federated information systems the processing is done on the up-to-date data in real time, without any preprocessing, differently from data warehouses (ETL). However, the performance of queries in data warehouses is usually better, since no query reformulation is required and organization of the data in a warehouse can be optimized for typical use of applications.

To summarize, data warehouses have the following advantages over federated databases.

1. High performance of queries once the data is loaded into a warehouse — no query reformulation is needed as in a federated database.

2. Processing historical data in addition to up-to-date operational data.
   Federated databases, for comparison, operate on the current data only.

Federated databases have the following advantages over data warehouses.

1. Initial processing of data, ETL, is required for data warehouses. It involves network utilization of transferring all the data in an enterprise into a data warehouse. In federated databases no initial processing is required.

2. All the transferred data to data warehouses is stored. No additional storage is needed for federated databases, except for processing temporary results and caching.

3. In data warehouses, data is not always up-to-date. In contrast to warehouses, in federated databases, data always reflect the latest updates.
4. It is impossible to update data via a data warehouse. In federated databases, data in the data sources can be updated, while preserving Global Serializability.

In both approaches, syntactic and semantic heterogeneity must be resolved. The data must be mapped to one common format and one common schema, using techniques such as schema mapping, described by Sheth and Larson [55]. The design proposed in [55] uses Common Data Language for integration of schemata of the data sources. The authors imply that the relational model is not well suited for data integration and some semantic model is needed:

“Database design and integration is a complex process involving not only the structure of the data stored in the databases but also the semantics (i.e., the meaning and use) of the data. Thus it is desirable to use a high-level, semantic data model for the Common Data Language...we believe that future systems are more likely to use a semantic data model or a combination of an object-oriented model and a semantic data model.” [55]

After integrating data from multiple sources using a Common Data Language, views on the integrated data can be created. The views can be in various formats different from the Common Data Language used for integration. For example, data from various relational data sources can be translated to RDF for integration. RDF in this example is the Common Data Language. Once the integration is done in RDF, relational or XML views can be defined over the integrated data.

The DOD example, described in a US Senate hearing [4], can illustrate the software complexity [41] and the costs involved in development of integration systems. The DOD invested more than half a milliard dollars in a system called DIMHRS—Defense Integrated Military Human Resources System. The system was intended to integrate with or subsume the 90 existing systems related to personnel and payroll in all the services of DOD. The DOD worked on the system from 1998 until the cancellation of the project in 2010. The DOD and its contractors failed to implement a working solution, so DIMHRS was never deployed.
2.3 Impedance Mismatch

The programming language we developed is intended to resolve the *impedance mismatch* between the Object Oriented model and the Semantic Web. In this section we explain what is impedance mismatch, using the discrepancies between OO and the Relational model as examples.

*Impedance mismatch*\(^3\) is a term used in the database community, for example in [58], to designate inherent discrepancies between models of popular programming languages and the Relational model. This term is used to describe the problems in development of applications for processing relational data that are caused by the discrepancies.

It is commonly considered that the Object Oriented (OO) model [47] is today the most popular model of programming languages.\(^4\)

Object Oriented and Relational are intrinsically different models. Table 2.1 shows artifacts in OO versus their counterparts (roughly speaking) in the Relational model.

<table>
<thead>
<tr>
<th></th>
<th>Relational</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>relations</td>
</tr>
<tr>
<td>objects</td>
<td>tuples</td>
</tr>
<tr>
<td>methods</td>
<td>queries, triggers and stored procedures</td>
</tr>
<tr>
<td>pointers/references</td>
<td>foreign keys</td>
</tr>
<tr>
<td>object identity</td>
<td>primary key (optional)</td>
</tr>
</tbody>
</table>

The most significant discrepancies between the OO model and the Relational model are as follows:

\(^3\)The term stems from *impedance matching* in electrical engineering and is related to *electrical impedance*. The term is related to opposition of an electric circuit to alternating current and is relevant to computer science only linguistically, giving name to the Object-Relational impedance mismatch.

\(^4\)For example, according to the TIOBE Programming Community Index of November 2011— [http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html](http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html), the Java programming language (which is an OO language) has the largest score among all programming languages. The category of Object Oriented languages has the largest score among all programming language categories. The index is calculated by counting hits in the most popular search engines, and is, to some extent, representative of the popularity of programming languages.
• Objects are transient. That is, they cease to exist after the end of the program execution. Tuples are persistent. In other words, they retain their values between different runs of programs processing them.

• There is a generalization (an inheritance or IS-A) relationship between OO classes. There is no such relationship between relations.

• OO methods are formulated in an imperative style, employing loops, if-else and other control flow statements, whereas queries are formulated in a declarative way.

• Objects in some OO languages such as C++ can be composed of other objects. Tuples are flat. In other words, tuples cannot contain other tuples as their values.

• Objects can contain references to other objects. Tuples can include foreign keys, implicitly referencing other tuples. However, such references only exist at the logical level and not at the physical level of the data.

• Objects have identity. References to objects can be created and objects can be accessed directly by these references. Tuples may have primary keys, so that other tuples can reference them implicitly by creating foreign keys. (The keys serve as constraints on values of attributes and do not provide direct access mechanism to tuples.)

All the differences described above mean that in order to program OO applications for processing relational data, translation between the models must be done. The applications must exchange the tuples into objects, work with these objects and later exchange the objects back into tuples. The programmers must work with two different models in order to program, understand and maintain these applications. This requires skills in two different models and increases the mental load on the programmers—making writing and maintaining applications harder. The Object-Relational Mapping [34] tools somewhat alleviate the translation between the two models, however the discrepancy still exists.

There exist different sets of tools for processing artifacts of OO and of the Relational model. Syntax checkers, refactoring and verification tools,
optimizers are tools for OO languages. Query engines and query optimizers are tools for relational data. The problem with working in two models simultaneously is that the tools of one model cannot process artifacts of the other model. For example, when SQL queries are embedded in programs as strings, syntax checkers cannot verify that the values of tuples are translated into correct types of the programming language. The syntax checkers are not aware of SQL and cannot check the validity of SQL syntax and the correct names of SQL columns. The same problem exists for refactoring tools. When, for example, the type of some variable in a program is changed, no such change can be automatically done in SQL queries to adjust the type of the corresponding attribute accordingly. The tools associated with the relational model cannot process artifacts of OO languages. For example there is no possibility to pose an SQL query on objects in an OO language. The LINQ [46] (Language Integrated Query) framework adds some limited query capabilities to the Microsoft .NET application framework.

Impedance mismatch, in general, designates discrepancies between the model of the programming language of an application and the model of the data that is processed by the application. For example, the ROX impedance mismatch [46] designates the discrepancies between the Relational, the OO and the XML models.

### 2.4 Persistent Programming Languages

The programming language we developed, is a persistent programming language [10]. In this section we discuss persistent programming languages and their benefits.

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

As an example of the work the programmer has to perform for explicitly loading and saving data, consider the work in Java with data items residing in relational databases by using the JDBC technology [53]. Consider a scenario in which the programmer has to update some value in a database. She has to write code for opening a JDBC connection to the database, creating an SQL statement, executing the statement, reading the result of the statement into a local variable, updating the variable, executing an
update statement, and, finally, closing the connection.

The Object Relational Mapping technologies, such as Java Persistence API [42], can alleviate this work by opening and closing database connections, creating and executing statements automatically, behind the scenes. However the programmer still has to specify explicitly when the mapped object is loaded, saved, or when its state is merged with the database, or is refreshed to reflect changes in the database, overriding the previous changes done to the object.

The Orthogonal Persistence Hypothesis 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.” [8]

Note that the transient-versus-persistent discrepancy between objects in OO and tuples in a relational database, is part of the Object-Relational Impedance Mismatch. This particular discrepancy is resolved in a persistent OO programming language.

2.5 The Ruby Programming Language

The programming language we developed is an extension of the Ruby programming language [33]. The Ruby programming language is a dynamic, object-oriented, interpreted programming language. It has high capabilities for meta-programming. In particular, a Ruby class may be written in several places in the code, and the methods can be dynamically added, removed, renamed (via the alias mechanism) in any place in the code. The language environment provides “hook methods” for various language events. For example, there are hooks that are called when a new method is added or when 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 parentheses after method calls are optional, makes Ruby a very powerful language for defining new Domain-Specific Languages on top of it.
Chapter 3

The SWOOM model

In this chapter we describe the Impedance Mismatch problem between the OO and the Semantic Web, we propose a model intended to solve the mismatch and we specify the principles behind the proposed model.

3.1 The OO-RDF Impedance Mismatch

In Section 2.3 we described the Impedance Mismatch between the Object Oriented model and the relational model. In a similar way there exists an Impedance Mismatch between OO and the Semantic Web data model (RDF, RDFS and OWL). In this section we provide a brief reminder of the Semantic Web data model and describe the main discrepancies with OO.

On the Semantic Web, the data items are *individuals*. Individuals have URIs and can appear as *subjects* or *values* in *subject-property-value* triples. When an individual appears in a triple \((s,p,v)\) as a subject, it is said that the individual \(s\) has property \(p\) with value \(v\). Individuals can be linked one to another by properties, forming a graph. Values of properties can be either individuals or regular data types, e.g. strings or integers. Individuals can belong to multiple classes. The class membership is either specified explicitly or deduced from properties of individuals. A subclass relationship (IS-A) can be specified for classes of individuals.

Knaublauch et al. [45] describe the differences between the Semantic Web and the OO model. The main discrepancies are as follows.

- **Differences in identity**: In the OO, every object has identity, which
is set during run time and is local. In the Semantic Web, every individual is identified by a globally unique permanent URI.

- **Persistence**: In the OO model, objects are transient. In the Semantic Web, individuals reside in some data source and are persistent.

- **Methods vs. Properties**: In the OO, objects can have methods and data fields. In the Semantic Web, there is no counterpart of methods—individuals have only properties.

- **Multiple vs. Single class membership**: In the OO model, every object is a member of a single direct (most specific) class. In the Semantic Web, an individual can be a member of multiple classes.

- **Dynamic class membership**: In the OO model, objects cannot change their class. In the Semantic Web, individuals can change their classes dynamically.

The following example illustrates the differences between the OO model and the Semantic Web model.

**Example 3.1**

Consider a person John, who is both a student and a chess player. Suppose that there is a need to write an application where the object representing John must be a member of two classes—*Student* and *ChessPlayer*. 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 construct a new class, say *StudentChessPlayer*, that implements interfaces *Student* and *ChessPlayer*. That is, in OO, a new class must be created for representing combination of classes. This leads to creating unnatural combinations, e.g. *StudentChessPlayerParentDriver*. Such solutions complicate the application by requiring maintenance 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*. 

25
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 ratings. Dynamically adding a property `hasRating` to an existing class `ChessPlayer` is straightforward in the Semantic Web, while in Java, the definition of the class `ChessPlayer` must be changed offline, and the class must be recompiled. (Some programming languages, for example Ruby [33], allow adding methods or fields dynamically. Yet, this does not solve the other issues of the impedance mismatch.) ■

3.2 Principles of SWOOM

In this section we propose a model, which is a hybrid between OO and RDF, for solving the OO-RDF Impedance Mismatch. We call this model *Semantic-Web Object Oriented Model*, abbreviated SWOOM. In the rest of the chapter we describe the principles of SWOOM and in the next chapter we describe an example implementation of SWOOM—an extension to the Ruby programming language that we call *Ruby on Semantic Web*.

The principles of SWOOM guarantee that objects will be *first-class citizens* in the Semantic Web and RDF individuals will be *first-class citizens* of OO programming languages. For each principle we provide a definition and rationale. For some of the principles we provide implementation notes and an illustrating example.

3.2.1 URIs as Language Literals

URIs are the basis of the Semantic Web. Every subject and every property in RDF triples has to be a URI. Note that the object of an RDF triple can be either a URI or a literal. The URI namespaces [21] are used in XML and in RDF. In RDF, they are used to save typing full URIs. When RDF data is exchanged, presented or queried, URI namespaces are defined at the beginning of a document or of a query, to save typing or presenting full URIs.

In Semantic Web representations, there is a way to assign a *prefix* to a URI namespace. Once the prefix is assigned, it can be concatenated with a *local name* to form a *qualified name* (*QName* in XML). Colon (:) is used...
to separate between the prefix and the local name. The qualified name
represents the full URI that is a concatenation of the URI namespace and
of the local name.

The following example illustrates a scenario of defining prefixes for URI
namespaces and using qualified names. In order to save typing a full URI,
such as `<http://www.acme.org/company#acme>`, a prefix company can be
assigned to the URI namespace `http://www.acme.org/company#`. The pre-
fix can be used later to form qualified names, such as `company:acme`. In
this qualified name, `company` is the prefix and `acme` is the local name. The
qualified name represents the URI `<http://www.acme.org/company#acme>`
that is a concatenation of the namespace assigned to the prefix and the local
name, in this case a concatenation of `http://www.acme.org/company#` and
`acme`.

**Definition**

URIs and qualified names are literals of the language in SWOOM. In other
words, they are written in the code directly rather than as strings.

**Rationale**

Since URIs are ubiquitous in Semantic Web representations, using them as
literals of the language facilitates programming. It eliminates the need for
typing quotes for strings, so the size of the program is reduced.

**Example**

The following pseudocode illustrates URIs as literals of a programming lan-
guage.

```
// passing individual
// represented by <http://www.acme.org/company#acme>
// as a parameter to the function foo()
foo(<http://www.acme.org/company#acme>);

namespace <http://www.acme.org/company#> company;
```

Listing 3.1: URIs as literals in a programming language
3.2.2 URIs and Access Transparency

Another principle of SWOOM is *URIs and Access Transparency*.

**Definition**

Every object has a URI, similar to RDF individuals. Any object of the language and any RDF individual can be accessed by a URI, providing *access transparency* [30]. The URIs provide a direct access to objects, as if they are data items in a database. A method in the language can either access an object directly by a reference to it (a pointer) or using a URI. For instance, a program may store a list of object URIs on a disk and later, read this list for accessing these objects. This way, a uniform access to all the items in SWOOM—both objects and RDF individuals—is provided. Access transparency means that the execution environment hides from the programmer how the item is accessed physically—whether it is retrieved from the memory, or from some data source, local or remote. From the programming language perspective, a URI serves as a global pointer to objects both in local memory and anywhere on the Web.

**Rationale**

The same code can be used to access all the artifacts in the programming language, so code duplication is reduced and code reuse is increased.

3.2.3 Location, Migration and Replication Transparency

The following principle is *Location, Migration and Replication Transparency*.

**Definition**

*Location transparency* [30] is the requirement that the location of data items is specified outside of the program code. *Migration and replication transparency* [30] are the requirements that the data items could be migrated or replicated without changing the code. Programmers can manipulate data
using only URIs, and they are relieved from specifying the locations of the data. A program in SWOOM is written as if it works over the entire Semantic Web, while the boundaries of this Web are defined by some mapping between URIs and data sources. The data sources and the mapping can be changed without changing anything in the application logic. Note that a URI, differently from a URL, is location independant. The data items can even be replicated in multiple sources for various reasons, for example for increasing resilience to failure. The fact that the data items are replicated should be hidden from the programmer and the replication should be done automatically by the execution environment.

Rationale

The benefit of this principle is that less code maintenance is required when the data sources or their location are changed.

Implementation Notes

Given an object, the decision regarding the location of the data, represented by the object, can be made based on the URI of the object.

3.2.4 Property Attachment

The next principle of SWOOM is *Property Attachment*.

Definition

RDF properties can be attached to objects. Note that properties can be added dynamically, during run time, not just during the construction of the objects. The objects of the language can be subjects in RDF triples or values of RDF properties. Once properties are attached to objects or once objects become values of properties of other objects or of RDF individuals, the objects can be queried by RDF query languages, such as SPARQL, similarly to regular Semantic Web individuals. Properties can be used to link objects with other objects, and with RDF individuals. This way objects can be added to the global Semantic Web.
Rationale

The same queries can be used for processing Semantic Web individuals and objects of the language, so duplication of queries will be reduced and reuse of queries will be increased.

3.2.5 Procedural Attachment

Next we present the principle of Procedural Attachment.

Definition

Every class, including RDF classes, can have methods. The methods are called on the instances of the class. All the methods have URIs, similar to properties in RDF, and they are serializable, conforming to an RDF format. Thus, it is possible to call methods explicitly using their URIs. The methods and the RDF properties are accessible in an identical form, i.e. using the same language notation.

Rationale

Procedural attachment is mentioned as a useful feature in [38]. Procedures written in an OO language would add general-purpose language capabilities, such as iteration, conditional constructs and arithmetic operations, to the declarative query languages SPARQL and SPARQL/Update. SPARQL lacks the notion of stored procedures. The only similar concept is an extension function. However extension functions can be used only in SPARQL FILTER and ORDER BY clauses. (FILTER clauses restrict solutions produced by SPARQL queries. ORDER BY clauses specify the order of the results.) There is no way in standard SPARQL to apply a function to the results of a query, or to a variable or RDF individual inside the query graph pattern.

On one hand, adding general purpose language capabilities to a declarative query language can significantly complicate query processing. On the other hand, such capabilities can be beneficial. In cases when the declarative capabilities of a query language do not suffice, query writers would be able to formulate their queries in an easier and more efficient way using
general purpose language constructs, such as condition statements or arithmetic operations. The same argument is valid for ontologies—while adding general purpose procedures to a declarative ontology language would complicate logical reasoning involving the procedures, in some cases using the procedures would be a more clear way to formulating some concepts than a pure declarative approach.

Example

As an example, consider a property `ont:age` for specifying the age of a person (here we use a qualified name with prefix `ont`). This property could be defined as a “procedure” that would return `currentDate() - ont:birthDate` in the programming language, where `currentDate()` is a method that returns the current date and `ont:birthDate` is a property that specifies the date of birth of a person. The property `ont:age` will be an equal citizen to other properties, so it can be queried using standard SPARQL syntax. For example, we can pose the following query:

```sparql
SELECT ?age WHERE { <www.example.com/persons/1> ont:age ?age }
```

The property `ont:age`, defined as a procedure, can also participate in OWL ontologies and can be subjected to reasoning. For example, a class `MinorCitizen` can be specified by a restriction on the `ont:age` property. (In OWL, classes can be specified by restricting properties of other classes.) The “procedure” `ont:age` will be called during execution of SPARQL queries and reasoners. SPARQL query processors and reasoners should be modified to run the code of the “procedural” properties. The “procedural” properties will be transparent to the programmers of queries or ontologies. The programmers could be oblivious to the fact that `ont:age` is a “procedural” property and will work with it as with an ordinary property.

3.2.6 Multiple Dynamic Class Membership

Another principle of SWOOM is Multiple Dynamic Class Membership.

Definition

`Multiple Dynamic Class Membership` means that every object can be a direct member of multiple classes, and the classes could be changed dynamically,
during runtime, as in the Semantic Web. We define direct class membership as follows:

**Definition 1.** An object \( o \) is a direct member of a class \( C \), if \( o \) is a member of \( C \) and \( o \) is not a member of any class that inherits from \( C \).

When an object is a member of multiple direct classes, the methods defined in all the direct classes can be called on the instance of the object. Also, when queried about its type, the object must return all its direct and indirect classes.

**Rationale**

The single-vs-multiple and static-vs-dynamic class membership discrepancies between OO and the Semantic Web will be resolved.

**Example**

Following Example 3.1, that illustrates multiple class membership, an object can directly belong to *Student* and *Employee* classes, while these two classes do not inherit one from the other. The classes *Student* and *Employee* can inherit from class *Person*, so the object will indirectly belong to class *Person*. For comparison, in OO languages, an object is usually a direct member of merely one class and an indirect member of all the classes the direct class inherits from.

Continuing with Example 3.1—an object can be a *ChessPlayer*, a *Student* and an *Employee*, in parallel, and can change its classes dynamically, for example ceasing to be *Student*.

**3.2.7 Orthogonal Persistence**

*Orthogonal Persistence* is another principle of SWOOM.

**Definition**

A programming language is *orthogonally persistent* [9] when persistence is implemented as an intrinsic property of the execution environment (see also Section 2.4).
Rationale

The programmers are not required to explicitly handle loading and storing data in the data sources. This will facilitate programming and reduce the amount of bugs.

Implementation notes

The execution environment can deduce according to the URIs of objects how to access the objects that are not in memory and how to reflect changes to the objects that are in memory. This means that the programmer should be relieved from the burden of explicitly loading and storing data items. She could work with the data items directly—by specifying their URIs and performing assignments, arithmetic and other operations directly on the data items by merely referring to their URIs.

Example

Suppose some company decides to replace some legacy database where it stores information about its employees. The company will just change the mapping from the URIs to the data sources, to specify that the data items representing employees from now on reside in the new database. Since the decision logic is separated from the program, no change in the program code will be required. Once the mapping is changed, the execution environment will seamlessly store/load information about employees to/from the new database.

3.2.8 Model Transparency

The next principle of SWOOM that we present is Model Transparency.

Definition

Model Transparency requires the programs to be written as if all the data being processed are in RDF, while the actual data (in different sources) is represented in different models and stored in different formats.
Rationale

The same code can be used for processing data in different models, so code duplication is reduced and code reuse is increased.

Implementation notes

In the language environment, multiple heterogeneous data sources can be connected via adapters. These adapters can create a virtual RDF view on the data in the data sources.

Example

Continuing the example from Subsection 3.2.7, suppose the company wants also to change the type of the database where it stores information about its employees, for example to replace a legacy hierarchical database by a relational database. The company will use an adapter from the relational model to RDF, for example the adapter presented by Bizer and Seaborne [19]. The company will change the mapping of URIs to data sources as in the previous example without changing anything in the code of the program. The actual underlying model of the data source is transparent to the program, so it does not matter to the program whether the model is changed. The program will continue to be able to process data about employees, as if these data are in RDF format. For example, the program could perform SPARQL queries about employees.

An interesting case is composing model transparency and replication—the data items can be replicated in different data sources in different formats. These differences should be hidden from the programmer—they should be specified in the mapping between the URIs of the objects and the data sources and should be transparently handled by the execution environment.

3.2.9 Persistence Independence

Now we present another principle of SWOOM—Persistence Independence.

Definition

According to Atkinson et al. [9], a beneficial design principle of a persistent programming language is persistence independence. The principle states
that the programmer should be oblivious to whether objects are persistent
or not, i.e. code should operate in the same manner on both persistent and
transient objects. This means that operations of the language such as access,
assignment and arithmetical operators, such as . (dot), = and + in Java,
should be performed both on the transient objects of the language and on
the persistent data items, without any syntactical difference.

Consequently, it must be possible to pose SPARQL queries over all the
objects of the language, both the in-memory objects and the ones represent-
ing persistent data. Posing queries over objects of a programming language
is somewhat similar to the LINQ technology [46]. In LINQ, SQL-like queries
can be posed merely over the native objects and arrays of a programming
language. Being able to pose SPARQL queries on all the objects of the lan-
guage makes these objects “equal citizens” in this aspect, and satisfies the
persistence independence principle.

Rationale

The same code can be used for processing persistent and transient entities,
so code duplication is reduced and code reuse is increased.

3.2.10 Language-Integrated Queries

The ability to integrate queries in the ordinary code of the programming
language is the next principle of SWOOM.

Definition

The principle of Language-Integrated Queries requires SPARQL and SPARQL/Update queries to be a part of the programming language.

Rationale

The queries should be language-integrated (first-class citizens of the lan-
guage), as in LINQ [46]. For comparison, consider how SQL queries are
written in Java using JDBC technology. The queries are written as strings,
so no error checking can be performed on them by the compiler. The Java
compiler is unaware of the syntax of the queries, so it cannot check, for
example, that column names are correct. To avoid code duplication in
SQL queries embedded in Java, string manipulation, such as concatenating strings and string variables must be performed.

In contrast with Java and JDBC, in a language, in which queries are language-integrated, the compiler is aware of the query structure, so it can perform error checking and optimizations involving the queries. Queries can be written directly as statements of the language and not in strings. Queries can contain regular variables of the language and can be integrated with other language constructs, such as loops and conditional statements.

3.2.11 Integrated Logical Inference

Next we present the principle of Integrated Logical Inference.

Definition

Integrated Logical Inference principle requires the language environment to infer new facts from explicitly stated facts, by applying logical reasoning, behind the scenes. The fact that the new information is inferred should be transparent to the code of the program—there should be no difference between the original data and the inferred data.

Rationale

Logical Inference is one of the important and powerful features of the Semantic Web, and it can be naturally applied in SWOOM. Using general-purpose logical reasoning together with declaratively specified logical rules could replace ad-hoc programming of the rules and reasoning in an OO programming language. This way the programmers will have to write, maintain and debug less code. It also provides better separation between logical rules and the procedural code of the programs.

Implementation Notes

An existing general-purpose Semantic-Web reasoner can be “plugged into” the language environment for inferring new information.

Example

The following example illustrates the power of inference.
Example 3.2

In this example, we will use names of people as names of variables. We will use ont as a namespace for properties and classes. The assignment operator (=) will specify assignment of values to properties, and the dot operator (.) will specify accessing a property of an individual.

Suppose one of the data sources contains ontological axioms related to a domain of family relations, which contains notions of relatives, siblings, parents, children etc. Suppose some of the axioms state that the properties ont:hasBrother and ont:hasSister are sub-properties of a property ont:hasSibling. The ranges of the properties ont:hasBrother and ont:hasSister are ont:Male and ont:Female, respectively. The property ont:hasSibling is symmetric and transitive.

Then, from a statement John.ont:hasSister = Mary in the code, the language environment can infer that Mary is a female and a sibling of John. In addition, John is a sibling of Mary (since ont:hasSibling is symmetric). In following statements, examining the John.ont:hasSibling property will return Mary. From the additional statement Mary.ont:hasSister = Alex, the language environment can infer that Alex is also a sibling of John (since the ont:hasSibling property is transitive).

As it can be seen, the language environment managed to infer new facts from the ones explicitly programmed in the code. Additional use of logic inference is validation of the data—the language environment can discover that the data produced by the program code is logically inconsistent. Suppose that one of the ontological axioms states that ont:Male and ont:Female are disjoint classes. In other words, there is no RDF individual that can be male and female at the same time. Now suppose the following erroneous statement is written in the program code: John.ont:hasBrother = Alex. A logical reasoner can infer from this statement that Alex is a male, however from the previous statement Mary.ont:hasSister = Alex it follows that Alex is a female. The reasoner will discover the inconsistency and could produce a runtime error, preventing inconsistent data from being added to the system. ■
3.2.12 Transaction Management

The final principle of SWOOM is the ability to specify transaction management directives in the code of a programming language.

Definition

Directives for transaction management must be part of the programming language in SWOOM.

Rationale

To enable correct updates of the persistent data and to provide the atomicity of the updates, transactions should be part of the language. The execution environment should be able to perform the transactions on the federated data sources while preserving Global Serializability. Global Serializability means that the united schedule of all the schedules in all the federated data sources is serializable. An algorithm for achieving global serializability is described by Raz [52].
Chapter 4

Ruby on Semantic Web

In this chapter we describe the programming language we developed based on the SWOOM model. We developed the language for demonstrating and evaluating programming in SWOOM. First we present the architecture of the language environment. Then we explain how the SWOOM principles were implemented in the language.

4.1 The Architecture

In this section we present the architecture of the language environment for the programming language we developed, based on our proposed hybrid OO-RDF model (SWOOM). The architecture is required to support the principles of SWOOM we presented in Section 3.2, such as location transparency, logical inference, orthogonal persistence, and others.

In order to demonstrate and evaluate programming in SWOOM, we implemented an extension of the Ruby programming language [33], as a an example implementation of the model. We called this extension and the execution environment to support it—Ruby on Semantic Web [28], abbreviated Ruby-S. We decided not to develop a new programming language from scratch, but to graft the desired features on an existing OO programming language, in our case, on Ruby. We chose Ruby since it is an Object-Oriented dynamic programming language with high metaprogramming capabilities and flexible syntax. Due to these features, Ruby is suitable for defining new programming languages without developing them from scratch. The new
programming languages can be built as Ruby extensions (under limitations of Ruby syntax). We would like to stress that our goal is to explore which benefits any OO programming language with the unified model could provide, not specifically a language based on Ruby. We built our prototype on top of ActiveRDF [49]—a Ruby library for manipulating RDF data, and we used implementation ideas from [49]. (See an explanation in the Introduction about the differences between ActiveRDF and Ruby-S.)

The execution environment for Ruby-S serves as both an execution environment for our proposed language and a federated information system (see Section 2.2). The federated data sources are connected via adapters to RDF. The architecture of the execution environment is depicted in Figure 4.1. The parts that we added appear hatched (Ruby on Semantic Web extension, Ruby on Semantic Web “DNS module”, In-Memory RDF Store of transient objects).

As depicted in Figure 4.1, the data reside in several data sources—in documents, in several relational databases, in an RDF Store and in some non-relational database. There are adapters for transforming SPARQL queries and SPARQL/Update statements into queries/statements suitable to each data source, for example to SQL in case of a relational database. We extended a popular RDB-to-RDF mapping platform, namely D2RQ [19], to handle SPARQL/Update statements. We called the extension D2RQ/Update [29]. The adapters provide virtual RDF views on all the data of the enterprise. (There is no adapter for the RDF Store, since it already contains RDF data.)
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. Ruby-S uses the ActiveRDF Query Engine to execute queries.

The execution environment includes a “DNS” module that maps URIs of the objects to data sources, in order to determine which data source new data should be written to. While reading values of properties of individuals and running SPARQL queries can be done on a collection of federated sources, without specifying in which data source each piece of information resides, writing properties and running SPARQL/Update statements requires specifying where the new pieces of information must be added. The “DNS” module serves the latter purpose—it specifies where each triple must be written based on the subject, property or value of the triple. The most natural way is to define rules based on the URIs of the subjects, so all the information about individuals with a certain URI prefix will be added to a single data source.

An additional data source—an in-memory RDF store—is used to represent and to query information about the native Ruby objects during runtime (queries on individuals of the Semantic Web are evaluated with respect to the original data sources). The type of every new Ruby object and all the RDF properties attached to it, are added to the in-memory RDF store during run time. Once an object is garbage-collected, the information on the collected object is removed from the store. This in-memory RDF store is an implementation detail of our extension and is used for enabling running SPARQL queries on the native Ruby objects/classes. This data source is coupled with the ordinary Ruby memory manager by using Ruby metaprogramming hooks. It contains information about the classes of the Ruby objects and values of RDF properties attached to the Ruby objects. This way, information about every Ruby object is split between the regular memory of the program (the values of the instance variables and metadata of the object) and the in-memory RDF dataset (the class of the object and the values of the properties). Only the information about the class of the object is duplicated.

Ruby-S provides an abstraction, in which there is no difference between native Ruby objects and Semantic Web individuals. The Semantic Web in-
individuals are not represented internally in Ruby during run time. Instead, temporary internal objects to represent the individuals are created lazily. These temporary objects exist only when the individuals they represent are being processed, e.g. when the program reads or updates the properties of the individuals. From the programmer’s point of view, it does not matter how the Semantic Web individuals are represented. The programmer has an illusion of processing Semantic Web individuals directly. (Using the reflection capabilities of Ruby could confuse the programmer, however the assumption is that if the programmers use reflection capabilities with our extension, they should be aware of the implementation of Ruby-S.)

Semantic-Web reasoners can be plugged into the in-memory RDF store and into the adapters of other data sources, in order to enable logical inference over the data. We connect an OWL reasoner to our in-memory RDF dataset, using an assembler of the Jena Semantic Web framework [61]. Jena assemblers allow building RDF datasets with various reasoners “plugged in”, so queries over a dataset with reasoners will return both the original and the logically inferred information.

The changes to the data are written immediately after every update (it is a possible optimization to group several updates and perform them together). We did not handle consistency and transactions issues, since they are not yet part of SPARQL/Update. However, we explain in Subsection 4.2.11 how transaction management can be implemented in Ruby-S.

An application that runs on top of Ruby-S is unaware of all the lower layers of the execution environment. The application works with URIs and is oblivious to where the data reside, what are the formats of the data, 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. All these provide orthogonal persistence, persistence independence, and also access, location, migration and replication transparency—as defined in Section 3.2.

The system and the programming language are demonstrated in a short video via the following link http://youtu.be/9jNfKtPBDvE.
4.2 The Implemented Features

In the following sections we describe how we implemented in Ruby-S the language features we proposed in Section 3.2. For each implemented feature we provide a code example. In the code examples, both the reserved words of the language and the reserved words we introduce in our extension are marked in bold font.

Note that in Ruby, the semicolons for terminating statements are used only when multiple statements appear on the same line. In other words, in contrast to languages such as C and Java, the semicolon terminating a statement at the end of a line is optional—it is usually omitted. The hash signs (#) designate comment lines.

As mentioned in Section 4.1, our goal was to graft Ruby-S features on the Ruby programming language, without changing the Ruby execution environment—the parser, the interpreter, etc. That is to say, we had to “simulate” Ruby-S features using Ruby metaprogramming capabilities. While the Ruby language can be extended or “metaprogrammed”, there are certain syntax limitations that cannot be changed by metaprogramming. In the examples we provide an outline of our implementation, explain the Ruby features we used and describe the choices we made.

4.2.1 URIs as Language Literals

According to the principle of URIs as language literals, introduced in Subsection 3.2.1, the language must enable writing expressions that contain a URI, such as \(<http://www.acme.org/company#acme>\), directly in the code, rather than as strings. We did not find a way to enable such possibility due to the limitations of the syntax of Ruby. The syntax of operator > cannot be changed in Ruby—operator > has to be followed by an operand. Remember that our goal was to add the new features to Ruby without changing its parser.

As an alternative, we implemented URIs as calls to the \texttt{uri} method (defined and implemented by us), with the URI string as a parameter.

A language that has URI namespaces as literals must include expressions for assigning prefixes to URI namespaces and using the prefixes to form qualified names. Assigning prefixes to URI namespaces must be performed by some keyword, e.g.
 namespace <http://www.acme.org/company#> company.

To enable assignment of prefixes, we introduced a new keyword, namespace, as an operator with two parameters: the URI namespace and the prefix. Listing 4.1 below illustrates usage of the new keywords namespace and uri.

Listing 4.1: Namespaces in Ruby-S

```
1  print uri 'http://www.acme.org/company#acme'
2
3  namespace 'http://www.acme.org/company#', :company
4  print company::acme
```

The code on Line 1 prints some representation of the data item with URI http://www.acme.org/company#acme, using the new keyword uri and the full URI. On Line 3, the prefix company is assigned to a URI namespace http://www.acme.org/company#. On Line 4, the prefix company is concatenated to a local name acme, with double colon (::) as a separator. The qualified name company::acme represents a concatenation of the URI namespace and the local name—the URI http://www.acme.org/company#acme. The code of Line 4 is equivalent to the code of Line 1.

Note that we used the double colon as a separator between the prefix and the local name. It was done in contrast to XML and RDF representations, where a single colon is used. We used the double colon due to the limitations of the syntax of Ruby, since there is no possibility to extend Ruby so that the single colon is used as the separator, without changing the parser of Ruby.

**Implementation**

The challenge in the implementation of the namespace feature is to introduce a new “keyword”—namespace—without changing the underlying language implementation, i.e. without rewriting the interpreter of the language. Introducing new “keywords” in Ruby is possible because of the following features:

- The parentheses after method calls in Ruby are optional. That is the code line namespace '...', :company is syntactically equivalent to the code line namespace('...', :company), i.e. a call to the method namespace with two parameters.
• The top-level method calls (not called from some other method) that are coded without specifying the “receiver” object, are implicitly executed on some predefined object. That is the line `namespace(...) is syntactically equivalent to the line `self.namespace(...)`. The keyword `self` in Ruby is similar to `this` in Java, and it designates the object in the context of which the call is executed. All the methods in Ruby are executed in the context of some object, with top-level method calls executed in the context of some predefined object.

The problem now is how to add the method `namespace` to all the possible implied objects. The solution is to add it to the `Object` class—the original class of Ruby, provided with the Ruby execution environment. It is similar to the `java.lang.Object` class in Java in the sense that all the classes in the language inherit from that class. So adding a method to the `Object` class would add the method to all the classes in the language. However, the `Object` class is a “system” class—it is provided by the Ruby execution environment, so how can it be changed without changing the Ruby execution environment? The answer is—in Ruby any “system class” can be changed like any other class. Note that in Ruby any class can be defined and redefined in several places as opposed to Java, for example, where a class definition must be fully specified in a single place. The methods of any class can be added and removed in several places. So the “system” classes in Ruby can be changed without changing the implementation of Ruby, in source files external to the Ruby “system” files.

We defined the `namespace` keyword as a method of the `Object` class. We implemented the method `namespace` to add assigned prefixes to a registry of namespace prefixes. Once prefixes are used to create qualified names, this registry is consulted to retrieve the corresponding URI namespaces.

The next challenge is to cause the execution environment to “understand” that the expression `company::acme` represents a qualified name for the URI `http://www.acme.org/company#acme`. Here we used the following syntax feature—in Ruby the double colon (::) is syntactically equivalent to the dot (.) operator, which is used to call methods on objects, in the same way as in Java. As we mentioned previously, the parentheses in Ruby are optional. Combining these two features of Ruby and the fact that all the methods in Ruby are executed on some implied object, the expression `company::acme` in Ruby is syntactically equivalent to the expres-
sion \( (\text{self}.\text{company}()).\text{acme}() \). (The first parentheses are not necessary—we wrote them here to stress the fact that the method \textit{acme} is called on the object returned by the method \textit{company}.) The problem now is that the \textit{company()} method is not defined in any class, so how could this non-existing method be executed? The solution is to define the method \textit{company()} dynamically during the call to the \textit{namespace()} method, i.e., to dynamically add this method to the \textit{Object} class. Note that in Ruby, methods can be added dynamically to any class and the addition can be done programmatically.

So \textit{self.company()} could be executed and some temporary object could be returned as a result, and on this object the method \textit{acme()} could be called. In case of the method \textit{company}, the \textit{company} identifier has been seen by the execution environment previously during the registration of the namespace (in Listing 4.1 on the third line) and can be added to the \textit{Object} class dynamically. However, in case of the method \textit{acme()}, it is never seen before it is called. So how can we cause the execution environment to execute such a non-existing method without having a chance to define the method before it is called? Here another feature of Ruby comes to our rescue—the \textit{method_missing} method. It is a special method in Ruby classes that is called by the Ruby execution environment every time a non-existing method is called on an object of the class. The name of the non-existing method is passed as a parameter. Thus the code \( (\text{self}.\text{company}()).\text{acme}() \) is equivalent to the code \( (\text{self}.\text{company}()).\text{method_missing}() \text{acme} \) in case \textit{acme} is not defined.

We decided to use the same mechanism of the \textit{method_missing} method to handle both prefixes (in this case \textit{company}) and local names (in this case \textit{acme}), for uniformity. We defined the \textit{method_missing} method of the \textit{Object} class to check if the method name is registered as a URI prefix in the registry of prefixes we mentioned earlier. In our case, the name \textit{company} was registered on the third line of Listing 4.1. The \textit{method_missing} method returns a temporary object that “remembers” the corresponding URI prefix, in our case \url{http://www.acme.org/company#}. The \textit{method_missing} method of that temporary object is defined to concatenate the prefix with the name of the missing method, in our case \textit{acme}, and to return an object that represents the data item that corresponds to the concatenated URI. In our case, the \textit{method_missing} of the temporary object concatenates
the URI namespace \texttt{http://www.acme.org/company#} with the local name \texttt{acme}, and returns an object that represents the data item whose URI is \texttt{http://www.acme.org/company#acme}.

Note that Line 4 of Listing 4.1 passes an RDF individual designated by the qualified name \texttt{company::acme} as a parameter to the method \texttt{print}. In Ruby-S, RDF individuals can be specified directly in the code, by using qualified names, can be passed as parameters to methods and can be assigned to variables.

### 4.2.2 Persistence Independence

The Persistence Independence principle, explained in Subsection 3.2.9, states that persistent and transient objects of the language are processed in the same way, applying the same language operators. In our case, it means that processing RDF individuals must be performed as processing of regular Ruby objects, i.e. by applying regular Ruby operators such as dot (.), assignment (=) and arithmetic operators. In the following paragraphs we provide examples of processing RDF individuals by regular Ruby operators and describe how we implemented these features.

#### Using the dot operator on RDF individuals

Listing 4.2 shows how the dot operator is used.

```
Listing 4.2: Accessing properties by the dot operator
1 namespace 'http://www.acme.org/company#', :company
2 namespace 'http://www.acme.org/ontology#', :ont
3
4 print company::acme.ont::profit
5 print company::acme.ont::manager.ont::salary
6 company::acme.ont::employee.each { |emp|
7   print emp.ont::salary
8 }
```

In Lines 1–2, the prefixes \texttt{company} and \texttt{ont} are defined. Line 4 prints the value of the property with URI \texttt{http://www.acme.org/ontology#profit} of the subject designated by URI \texttt{http://www.acme.org/company#acme} (recall the \texttt{subject-property-value} data representation of RDF). Here the dot (.)
operator of Ruby is used to access the values of the properties, in the same way as in access to methods of objects. The statement is translated into the SPARQL query shown in Listing 4.3

Listing 4.3: The SPARQL query performed by the execution environment for Line 4 of Listing 4.2

```sparql
SELECT ?var WHERE {
}
```

The query is executed by the Ruby-S execution environment behind the scenes, using the ActiveRDF Query Engine.

**Implementation**

The challenge here is to translate the statement `company::acme.ont::profit` into the SPARQL query shown in Listing 4.3—the execution environment must understand the URI of the subject and the URI of the property. Since both dot (.) and double colon (::) in Ruby are used for calling methods, the expression `company::acme.ont::profit` is syntactically equivalent to the expression `self.company().acme().ont().profit()`. Here we used the same technique as for handling namespaces, by using the `method_missing` method of the temporary objects returned by executing (non-existing) methods `company, acme, ont` and `profit`. A chain of calls to `method_missing` methods is performed with symbols `company, acme, ont` and `profit` as a parameter. The last `method_missing` method in the chain of calls, called with the `profit` symbol as a parameter, executes the query and returns the result.

An issue here is the cardinality of the result, i.e., should the returned object represent a single value or should it be a list of values (in Ruby there is no declaration of the type of returned values). Note that consecutive methods can be called on the returned object and properties of the returned object can be accessed. For example, on Line 5 of Listing 4.2, the property `ont::salary` of the value of the property `ont::manager` of the subject `company::acme` is accessed. Remember that in RDF, a value of a property of some subject, can be a subject in other triples and as such itself can have properties. Also note that in RDF unless a property is known to be func-
tional according to some ontology, there could be multiple triples with the same subject and the same property, but with different values. That is to say, an RDF individual can have multiple values for the same property. On Line 6 of Listing 4.2, the method *each* is called on the value of the property `ont::employee`, meaning that this value is *Enumerable*—it is a list of values.

So, the problem is that the method that returns an object cannot anticipate how the returned object will be used—as a single value or as a list of values. We handled the problem in the following way—the Ruby-S execution environment consults the data sources if the property is known to be functional. If the property is known to be functional, a single Ruby object representing the result is returned. If the property is not known to be functional, an array of values is returned. In Listing 4.2 and in the following listings we assume that the property `ont::employee` is non-functional (i.e. the company can have many employees). All other properties are functional.

In case when no values exist for the property, if the property is functional, `nil` is returned (the counterpart of `null` in Java), otherwise an empty Array is returned.

The type of the returned object (Numeric, String etc.) is defined by the ActiveRDF query engine, using either the attached type of the RDF literal, or using heuristics to deduce the type. (Recall that Ruby-S is built on top of ActiveRDF.) Note that for the ActiveRDF query engine a problem exists when the result is a single value. ActiveRDF query engine must be instructed specifically what to do in such case—to return a single value or to return a list consisting of that single value. In Ruby-S, in contrast to ActiveRDF, the decision is based on whether the property is functional.

Note the loop construct of Ruby applied on Lines 6–8 of Listing 4.2—there is an iteration on the values of the list of employees, returned by accessing the property `ont::employee` of the subject `company::acme`. The loop iterates over the employees of the company and prints the salary of each of them. The iteration variable `emp` is used to represent RDF individuals designating the employees of the company. The property `ont::salary` is accessed through this variable. What we see here is an integration of RDF individuals and RDF properties with loops and variables of Ruby—RDF individuals are processed as regular Ruby objects.
Applying assignment operator on properties of RDF individuals

The following example illustrates assignment of values to properties of RDF individuals.

Listing 4.4: Properties and assignment

```ruby
namespace 'http://www.acme.org/company#', :company
namespace 'http://www.acme.org/ontology#', :ont

company::acme.ont::profit = 1000
company::acme.ont::manager = company::john
company::acme.ont::employee = [company::john,
                              company::mary]

# runtime error — assigning an array to
# a functional property
company::acme.ont::profit = [1000]

# runtime error — assigning a single value to
# a non-functional property
company::acme.ont::employee = company::jim

company::acme.ont::manager = nil
company::acme.ont::employee = []
```

Assignment in Ruby is implemented by using methods with `=` postfix. In case of Listing 4.4, these are the `profit=`, `manager=` and `employee=` methods (sic! The assignment sign `=` is part of method names). Note that the assignment operator can be separated from the names of the properties being set, despite that, the methods with postfix `=` are called. In case of the assignment on Line 4, it is syntactically equivalent to `self.company().acme().ont().method_missing('profit=',1000)`, i.e. the name `profit=` and the assigned value (1000) are passed as parameters to the `method_missing` method.

Implementation

We implemented the assignment operation using SPARQL/Update, by deleting the previous values of the property and inserting the new value. Here,
again, we have an issue of cardinality—should a single value or a list of values be assigned. We provided different semantics of the assignment, depending on whether the property is functional. For a functional property, only a single value can be assigned, as shown on Lines 4 and 5 of Listing 4.4. For a property that is not known to be functional, only arrays of values can be assigned, as shown on Lines 6–7. Lines 9–14 show the runtime errors that are returned by the Ruby-S execution environment when an array is assigned where a single value is expected, and vice versa.

Deleting values of properties of RDF individuals

For deleting values of a properties, we used the same semantics as we used for designating absence of values for a property—nil for a functional property and empty array for a non-functional property. See Lines 16–17 of Listing 4.4 for an example of how values of functional and non-functional properties can be deleted in Ruby-S.

Applying arithmetic operators on properties of RDF individuals

Listing 4.5 illustrates applying arithmetic operators on the values of properties.

Listing 4.5: Properties and arithmetic operators

```ruby
namespace 'http://www.acme.org/company#', :company
namespace 'http://www.acme.org/ontology#', :ont

print (company :: acme.ont :: profit +
      company :: newacme.ont :: profit)

company :: acme.ont :: profit += 100

print (company :: acme.ont :: employee +
      company :: newacme.ont :: employee)

company :: acme.ont :: employee += [company :: jim]
```

Lines 4–5 print the sum of the profits of the companies acme and newacme. Line 7 increases the value of the profit of acme by 100. Lines 9–10 print
the combined list of employees of the companies acme and newacme. Line 12 adds a new employee—company::jim—to the list of employees of the company acme.

Note the different semantics of the operator + when applied to properties ont::profit and ont::employee. It is a sum of numbers for the property ont::profit and it is a concatenation of lists for the property ont::employee. Here, again, the differentiation is done according to the property being functional. The property ont::profit is functional, so a single value is returned and the operator + has scalar semantics. The property ont::employee is non-functional, so a list is returned and the operator + has list semantics (in Ruby, the operator + is defined to operate on lists as well as on numbers).

To summarize, the Persistence Independence principle is manifested in Ruby-S by the fact that RDF individuals (persistent entities) behave as regular language objects—they can be manipulated by regular language operators, such as dot, assignment and plus. RDF individuals can be assigned to variables, passed to methods, can participate in loops, assignment, arithmetic operations and other expressions of the language.

### 4.2.3 Language-Integrated Queries

The principle of Language-Integrated Queries, explained in Subsection 3.2.10, states that SPARQL queries should be written in Ruby-S as is, rather than inside strings that are passed to a query engine for execution. The principle also states that the queries must also be applicable to regular objects of the language. In this subsection we illustrate how general SPARQL queries are embedded in Ruby-S. In Subsection 4.2.7 we will show how SPARQL queries can be posed on regular Ruby objects.

The challenge with embedding SPARQL queries in Ruby is how to introduce the SPARQL keywords and the structure of SPARQL queries into the Ruby programming language, and let the execution environment “understand” the queries, without changing the language itself.

Lines 3–5 of Listing 4.6 show how we integrated SPARQL queries in Ruby-S. The equivalent query in the original SPARQL syntax is shown in Listing 4.7.

Listing 4.6: A SPARQL query embedded in Ruby-S

```ruby
namespace 'http://www.acme.org/ontology#', :ont
```
Listing 4.7: The SPARQL query in the original SPARQL syntax, that corresponds to the query embedded in Ruby-S in Listing 4.6

```sql
PREFIX ont: 'http://www.acme.org/ontology#'

SELECT ?emp WHERE {
  ?emp ont:deptNo 1
}
```

Note that the only differences between the original SPARQL syntax and the syntax of SPARQL queries in Ruby-S, are using of the prefix `q::` to designate SPARQL variables (`q::emp` for `?emp`) and a double colon for concatenating prefixes and local names. The usage of double colon was explained in Subsection 4.2.1. The usage of the prefix `q::` is due to a limitation of the Ruby syntax—with the question mark the Ruby code would become invalid. Otherwise, the SPARQL query is almost identical—it is written directly in the Ruby code.

Ruby variables can be used inside queries, directly, without any string manipulation, differently from the case when queries are represented as strings. Listing 4.8 shows the same code as in Listing 4.6, while the only difference is that the variable `deptNumber` is used inside the query.

Listing 4.8: A SPARQL query with a variable, embedded in Ruby-S

```ruby
namespace 'http://www.acme.org/ontology#', :ont

departmentNumber = 1

SELECT q::emp WHERE {
  q::emp ont::deptNo departmentNumber
}
```

SPARQL queries in Ruby-S can be directly integrated with Ruby loops, as shown in Listing 4.9.
In Listing 4.9, a SPARQL query is integrated with the loop on Lines 6–8. The query returns the RDF individuals representing employees of Department 1. The loop is executed on the results of the query. The loop increases the value of the property `ont::salary` of each employee by 100.

**Implementation**

In the implementation of this feature we had to introduce a new construct—`select...where { ... }`, which includes a block `{}`). As explained earlier, we wanted to do so without changing the parser and the interpreter of Ruby. While in other languages introducing a new kind of block expression would require changing the language itself, in Ruby the block is a *first-class citizen*—it can be passed as a parameter to methods. That is, the expression `where { ... }` is syntactically equivalent to passing the block to the method `where`, i.e. `where({ ... })`. So, we can define a new method `where`, which receives a block as a parameter, and this way introduce a new block expression.

While the block issue is resolved, the challenge remains to introduce a whole new construct with two new keywords, a variable list between `select` and `where`, and a triple pattern list inside the block. To exemplify more complicated SPARQL queries, see Listing 4.10. It shows multiple SPARQL variables in the variable list and multiple triples in the triple pattern.

Listing 4.10: A SPARQL query with multiple variables and multiple triples in the triple pattern

```sparql
select q::name q::salary where {
  q::emp ont::deptNo 1;
}
```
In implementing the embedding of SPARQL queries, we used the previously mentioned features of Ruby:

- When the receiver of a method is not specified, some implied object serves as the receiver.
- There is a possibility to add methods to the Object class, and by that add methods to all the objects in Ruby.
- The parentheses after method calls are optional.
- The double colon (::) has the same meaning as the dot (.) operator.
- Blocks can be passed as parameters to methods and can be executed by these methods.
- The method_missing method is called when a non-existing method is invoked.

According to the features mentioned above, the code in Listing 4.6 is equivalent to the code in Listing 4.11. (We show both versions in Listing 4.11.)

Listing 4.11: SPARQL query embedded in the language and the equivalent version with parentheses and receivers specified

```ruby
namespace 'http://www.acme.org/ontology#', :ont

# the embedded SPARQL query
select q::emp where {
  q::emp ont::deptNo 1
}

# the equivalent code, with method calls, parentheses and receivers specified
self.select(self.q().emp(self.where({
  self.q().emp(self.ont().deptNo(1))
}))
```

```ruby
ont::name q::name.
q::emp ont::salary q::salary
}
```
To translate the code above to the original SPARQL, the sequence of methods shown on Lines 10–12 of Listing 4.11 must be executed, while the first method, `select`, has to execute the SPARQL query and return the result. The idea here was to add methods `select`, `q` and `where` to the Object class, and to define a set of temporary objects representing different states of parsing. Each temporary object “remembers” the parsing state of the query and passes that state to the next method in the execution sequence (from right to left, according to the sequence of the method calls). The next called method adds the parsing state of its parameter to the state of the receiver (another temporary object), and returns the result with the combined state. For example, the method `deptNo` on Line 11 learns that the parsing state (from right to left) is such that the constant 1 is encountered. The method `deptNo` returns a temporary object that represents the combined parsing state (from right to left)—the query fragment `ont::deptNo 1` is encountered. The method `emp()` receives the temporary object returned from the method `deptNo` and returns the object that represents the state of the parsed query—`q::emp ont::deptNo 1`. This way the query code “parses” itself from right to left. It executes the query keywords, variables, prefixes and local names as (either missing or defined) methods. During the execution of the “parsing”, temporary objects representing the parsing state are passed between the methods as parameters or return values. The last method (from right to left), `select`, receives a temporary object that holds the whole query “parsed”. The `select` method passes the query to the ActiveRDF query engine for execution and returns the results of the query.

### 4.2.4 Location and Migration Transparency

The Location and Migration Transparency principle, explained in Subsection 3.2.3, means that the locations of data items must be transparent to the code and independent from the code, i.e. the data processing code would not specify where the data items reside. To provide location and migration transparency, there is a need for some mechanism, external to the processing code, to specify where the data items are located. First, the SPARQL endpoints to send queries have to be specified somewhere. Secondly, for creating or updating data, simply specifying multiple federated data sources is
not enough. The execution environment should know to which specific data
source it must send new triples. The most natural choice would be to use
the URIs of the subjects or the URIs of the properties that appear in the
triples. Since URIs are unique and uniform, it is convenient to map the data
items to the data sources they should be stored in.

Semantic Web standards do not specify the relation between the URIs
of data items and the SPARQL endpoints that represent the data sources
of the items. Note that URIs of data items are not necessarily URLs. In
other words, given a URI, it is not clear where is the location of the data
item represented by the URI. In the Linked Data [13] implementation of
the Semantic Web, the URIs of the data items must be dereferenceable—
some representation of the data item must be returned when the URI is
referenced. However, there is no such requirement in the Semantic Web. In
addition, even in Linked Data there is no specification how to update the
data items according to their URIs. There exists a W3C draft for describing
RDF datasets—VoID (the Vocabulary of Interlinked Datasets)[6]. However
it has not yet reached the status of a standard. This draft specifies how
RDF datasets should be described, including the patterns of the URIs in
the dataset and the URIs of the SPARQL endpoints for the dataset.

Since no standard of mapping URIs to SPARQL endpoints exists, we in-
troduced a module that we called “DNS” (using an analogy with the Domain
Name System—a system that translates domain names into IP addresses).
This module maps URIs of data items to SPARQL endpoints of the data
source where the data items reside.

Listing 4.12 shows an example of “DNS” rules, also specified in Ruby.

Listing 4.12: “DNS” specification

```
1 source( :type => :sparql ,
3 :engine => :joseki , :results => :sparql_xml)
4
5 targetForProperty %r(http://www.acme.com/vocab/resource/projID) =>
6 'http://sw.cs.technion.ac.il:2020/SDB/update'
7
8 targetForSubject %r(http://www.acme.com/employees) =>
9```
The “DNS” rules that appear in Listing 4.12, specify a SPARQL endpoint for running SPARQL queries on the data and three SPARQL/Update endpoints for writing new information. The rules specify which triples should be written to which SPARQL/Update endpoint according to the URIs of the subjects or the URIs of the properties in the triples. The URIs are matched using regular expressions. The execution environment consults “DNS” when it executes SPARQL queries or SPARQL/Update statements. This is done behind the scenes, without interference of the programmer who writes the data processing code.

“DNS” provides location transparency since the programmer who writes the code of an application in Ruby-S is unaware of where the data items come from and where they are written to. The “DNS” rules are written once, they are concentrated in one place and separated from the data processing code of applications. “DNS” provides migration transparency, i.e. once the data source of some data items is changed, it is required only to modify the rule in “DNS” that specifies the SPARQL/Update endpoint for the migrated data items. The code of the application will remain unchanged and will use the same URIs as prior to the data migration.

4.2.5 Integrated Logical Inference

The Integrated Logical Inference feature of the language, explained in Subsection 3.2.11, means that new facts can be inferred from existing facts, in a transparent way, without explicitly invoking a reasoning engine. The program should work in the same way with inferred data, as it works with the original data. That is, the program should access the inferred properties of RDF individuals in the same way as non-inferred properties. Consider for example an ontology about a commercial company that defines the following axiom: if a department has somebody as a manager, then this person manages that department. In OWL such axiom can be expressed by a triple ont:hasManager owl:inverseOf ont:manages, where ont is some namespace used in the ontology and owl is the namespace of OWL. Now, having this
ontological axiom and a triple \texttt{dept::id2 ont:hasManager emp::id3} in one of the data sources, the code statements in Listing 4.13 will produce results including the inferred facts.

Listing 4.13: Example of accessing an inferred property

```typescript
1 namespace 'http://www.acme.org/ontology#', :ont
2 namespace 'http://www.acme.org/departments#', :dept
3 namespace 'http://www.acme.org/employees#', :emp
4
5 print dept::id2 . ont::hasManager # emp::id3 is printed
6 print emp::id3 . ont::manages # dept::id2 is printed
```

Line 5 of Listing 4.13 prints \texttt{emp::id3}, simply according to the data. Line 6 prints \texttt{dept::id3} despite the fact that this information does not exist in the data sources, but is inferred. The inference engine, plugged into the Ruby-S environment, infers the triple \texttt{emp::id3 ont::manages dept::id2}, and the result of the expression on Line 6 is produced. Note that the code in Listing 4.13 is agnostic to the fact that the calculation on Line 6 involved inference, i.e. the code is written as if a regular property access of an RDF individual is performed. Accessing both properties that appear in the data (on Line 5) and the properties inferred from the data (on Line 6) is performed in the same way. The reasoning occurs behind the scenes, automatically, without any involvement of the programmer of the code. The programmer is unable to distinguish between the data that originally exist in the data sources and the inferred data.

This feature can be enabled by plugging a reasoner into a data source for which inference is desired, and registering the data source with the plugged-in reasoner in “DNS” of the Ruby-S environment. SPARQL queries performed over a data source with a reasoner, produce results based on both the original and the inferred data. This usage of standard tools, such as reasoners, is one of the advantages of the Semantic Web. Since the Semantic Web is based on standards and formal logic, standard logic-based tools for processing Semantic Web data exist (some of them are open-source) and can be used.
4.2.6 URIs and Access Transparency

The URIs and Access Transparency principle, explained in Subsection 3.2.2, states that every object of the language should have a URI and should be accessible by this URI. Each RDF individual has a URI by definition. For Ruby objects, the questions are what should be their URI, how to assign URIs to the objects and how to access the objects according to the assigned URIs. When we implemented this feature, we wanted that the original Ruby code would become valid Ruby-S code without any change. We wanted every Ruby object to have a URI, including the Ruby objects created in the original Ruby code. Since the original Ruby code has no notion of URI, the URIs have to be added to the objects implicitly, behind the scenes. Every “new-born” object has to have a URI automatically, without requiring the programmer to specify the URI.

In Ruby, the implicit unique feature of every object is its object id. Every object has a method object_id that returns the id. The object can also be accessed by the object id using the _id2ref method of the ObjectSpace class of Ruby. So it was a natural choice to use this feature of Ruby for the implementation of object URIs.

We added to every object a getURI method that translates the ids of the objects to URIs of the form http://localhost/#<object id> (object id string concatenated with the prefix http://localhost/#). This way every object automatically has a URI immediately after its creation, including the objects of the original Ruby code. We changed the code that accesses RDF individuals by their URIs to consider the URIs with the prefix http://localhost/# as a special case and to retrieve them using the _id2ref method of ObjectSpace. By that, every object can be accessed by its URI, in the same programmatical way as the RDF individuals.

An advantage of adding URIs to the objects is that now the objects, similarly to the RDF individuals, can be persisted in one of the data sources, by specifying a rule in “DNS” (the module that maps URIs to data sources). The objects can be persisted, migrated and even replicated, in a way transparent to the programmer, without changing the code of the program, i.e. only by changing rules in “DNS”. This way, Location, Migration and Replication transparency (see Subsection 3.2.3) is provided to the original

\[^1\text{http://www.ruby-doc.org/core-1.8.7/ObjectSpace.html}\]
Ruby objects, similarly to the RDF individuals.

An additional advantage of adding URIs to the objects is that having URIs is the basic requirement to become a first-class citizen of the Semantic Web. The objects now can be subjects of RDF triples (as described in the next paragraph), can participate in SPARQL queries and can be subjected to logical inference.

4.2.7 Property Attachment

After adding URIs to the objects, the next step to make the objects first-class citizens of the Semantic Web is to enable adding RDF properties to the objects (See the Property Attachment principle explained in Subsection 3.2.4). Once the objects have properties, their properties can be accessed as the properties of RDF individuals and the objects can participate in SPARQL queries. Our implementation of property attachment and of running SPARQL queries on original Ruby objects is shown in Listing 4.14.

Listing 4.14: Property attachment to Ruby objects

```ruby
namespace 'http://www.acme.org/ontology#', :ont
class ChessPlayer
  def initialize(name, rank)
    @name = name
    @rank = rank
  end

  attr_accessor :name, :rank
end

p1 = ChessPlayer.new('John', 2)
p2 = ChessPlayer.new('Jane', 1)

print 'all the chess players:
print select q::player where {
  q::player rdf::type ChessPlayer
}
p1.ont::deptNo = 1
```
In Listing 4.14, a regular class of Ruby—ChessPlayer—is defined on Lines 2–6. Two objects of this class, p1 and p2, are created on Lines 11–12. The query on Lines 15–17 returns all the objects that are known to be of the class ChessPlayer. (For this property we use the rdf:type property that is part of RDFS.) Note the use of the name of a Ruby class (ChessPlayer), as is, in the SPARQL query. As a result of executing this query, both p1 and p2 are printed.

Line 19 illustrates adding the property ont:deptNo with value 1 to the regular Ruby object p1. (It is a property defined in an example ontology we created. It designates the number of the department an employee belongs to.) The property is added by using the dot (.) operator in the same way as properties of RDF individuals are set in Ruby-S.
Once a property is added to an object, its value can be accessed by using the dot operator in the same way as properties of RDF individuals are accessed. For example, the code on Line 21 prints the value of the property ont::deptNo of the object \textit{p1}. Compare how the rank of a \textit{ChessPlayer} is accessed according to the original Ruby syntax on Line 22. Here the method \textit{rank} of \textit{p1} is called. Printing the property \textit{ont:deptNo} is programmed by using the dot operator similarly to printing the rank of a \textit{ChessPlayer} by accessing the \textit{rank} method.

A more interesting case is using values of properties in SPARQL queries executed on the original Ruby objects. The query on Lines 25–28 returns all the objects representing the Ruby class \textit{ChessPlayer} that have the property \textit{ont:deptNo} with value 1. The query on Lines 32–35 is integrated with a Ruby loop construct—the method \textit{rank} is called for every object returned by the SPARQL query. As a result, the ranks of the chess players in the first department are printed. This shows how a SPARQL query, embedded in Ruby-S, is executed on Ruby objects with RDF properties attached to them, and how Ruby methods are executed over the results of the SPARQL query in a loop. This exemplifies how Ruby constructs—objects, classes, loops and methods—are integrated and freely mixed with Semantic Web constructs, such as RDF properties and SPARQL queries.

More intricate examples appear on Lines 37–42. The statement on Line 37 assigns the Ruby object \textit{p1} as a value of the property \textit{ont::hasManager} of the RDF individual \textit{dept::id1}. On Line 38, the value of the property \textit{ont::manages} is inferred by the Ruby-S execution environment in the same way as it is done in Listing 4.13. The query on Lines 40–42 returns all the subjects that have property \textit{ont:deptNo} with value 1, while the returned subjects can be both RDF individuals and Ruby objects. These examples show that Ruby objects and RDF individuals can be freely mixed in the language statements, can be linked together by RDF properties and can be subjected to logical inference.

**Implementation**

The implementation issue here is where to store the RDF properties of the objects—in the memory block allocated to each object or in some other data structure. If the properties would be stored in the memory block allocated to each object or in a special data structure, the federated query
engine would have to be made aware of these additional sources of data. In order to enable attaching properties to the objects without changing the federated query engine, we implemented adding properties of the objects to an additional RDF data source. This data source is added to the list of the federated data sources. This way, the properties of the objects will be queried by the federation query engine, transparently, as if they reside in some regular data source. So, we added a special in-memory RDF store to the Ruby-S execution environment to preserve the information about the objects, their types and their properties. Once a new Ruby object is created, information about it and about its Ruby type is added to the in-memory RDF store. Once the object is garbage-collected, the information is removed automatically. Note that a rule can be added to “DNS” in order to store some objects in some additional data sources, so objects could become persistent, transparently to the programmer, by changing the “DNS” rules, without changing the code.

The implementation challenge here is how to change the behavior of all the Ruby objects to be added automatically to the in-memory RDF store and also to be removed from it automatically after being garbage-collected. In Ruby, to create an object, the new method of the class of the object is called. For example, in the code

```
p = Person.new("John", "Doe")
```

the method new is invoked on the class Person. Every class of Ruby is an object of the class Class. Hence Person is an object of the class Class, so the method new defined in the class Class is invoked.

If we change the new method of the Ruby class Class, we can change the creation procedure of all the objects in Ruby. So, how can we change an existing method in Ruby? Remember that in Ruby any method of any class, including the “System” classes, such as Class, can be redefined dynamically, by defining a new version of the same method. In our case, we do not want to change the method completely, but to add some functionality to it, similar to adding an aspect to a method in Aspect Oriented Programming [43]. New functionality can be added to an existing method in Ruby by assigning an additional name to the original method (by using the alias keyword), overriding the original version of the method with a new method definition, and letting the new method call the old method by the alias. For example, if we have a method foo, we can create an alias originalFoo to the method foo, and after that create a new version of foo that will call the original version...
by calling originalFoo. So, we created a new version of the new method of the Ruby class Class, that adds information about the newly created object to the in-memory RDF store, after calling the original version of the new method.

To enable removing objects from the in-memory RDF store, we used the finalizer feature of Ruby. In Ruby, for every object a special method (finalizer) can be registered and this method will be called by the Ruby execution environment, before the object is garbage-collected. Thus, we added registration of a finalizer to our version of the method new in the class Class. This finalizer removes the object from the in-memory RDF store once the object is garbage-collected.

Assigning URIs to objects

Additional feature that we implemented is the possibility to assign ad-hoc URIs to objects, in addition to the automatically generated ones, as explained in Subsection 4.2.6. We present this feature in the Property Attachment subsection to illustrate how properties attached to an object can be accessed via the assigned URIs.

Listing 4.15 shows an additional feature that we implemented—assigning URIs to objects.

Listing 4.15: Assigning a URI to a Ruby object

```ruby
1 p1 = ChessPlayer.new('John', 2)
2 emp::id1 = p1
3 print emp::id1.rank
4 emp::id1.rank = 4
5 print p1.rank
```

In Listing 4.15, on Line 1 a Ruby object of the class ChessPlayer (from Listing 4.14) is created. The object represents a chess player with name John. On Line 2, there is an assignment of the object referenced by p1 to the URI emp::id1. Once the URI is assigned, the object methods can be called both via the URI, as on Lines 3 and 4 of Listing 4.15, and via a reference to the object, as on Line 5 of Listing 4.15. The URI serves as an additional reference to the object. The URI continues to reference the object until it is reassigned to another object or to nil. If p1 is reassigned to
another object, the URI will continue to reference the object that represents the chess player John. (Here the regular semantics of reference assignment in OO is applied.)

Implementation

In the implementation we used the OWL property owl:sameAs that allows to state that two RDF individuals are the same—\((x \text{ owl:sameAs } y)\). When such triple is presented to an OWL reasoner, the reasoner will infer new properties for both individuals, based on the properties of each of them. For example, if there exist triples \((x \text{ property1 value1})\) and \((y \text{ property2 value2})\), the reasoner will infer triples \((y \text{ property1 value1})\) and \((x \text{ property2 value2})\). After the inference is performed, both \(x\) and \(y\) will share the same properties.

We implemented the assignment of RDF individuals one to another, i.e. \(x = y\), by adding a triple \(x \text{ owl:sameAs } y\) to one of the data sources.\(^2\) After the assignment is done, \(x\) and \(y\) become identical for queries and inference. That is, they become the same entity, since the OWL reasoner plugged into our environment considers the \(owl:same\) property during its operation. In other words, \(x\) and \(y\) share the same properties after the assignment.

The challenge was to simulate this behavior in Ruby-S by allowing to call methods of the object via the URI it is assigned to—in our case calling the method rank on the URI emp::id1.

We implemented calling methods of objects via URIs by letting the method_missing method of temporary objects that represent RDF individuals to check if the RDF individuals have owl:sameAs property with a value of some object. Remember that we used the method_missing method to implement accessing properties of RDF individuals in Subsection 4.2.2. In the case of Line 3 in Listing 4.15, the method_missing method of the object representing emp::id1 is called with a symbol ‘rank’ as a parameter. The method_missing method checks if RDF individual with the URI emp::id1 is the same (according to the owl:sameAs property) as some other object that has a ‘rank’ method. If such object is found, its method ‘rank’ is called. If no such object is found, the method_missing method assumes that ‘rank’ here designates a URI prefix and is part of an expression involving accessing a

\(^2\)It is natural to give such semantics to the ‘=’ operator, since it has this semantics in the N3 RDF notation [15].
property, such as `emp::id1.rank::some_localname`. Since no namespace with the ‘rank’ prefix was registered previously, a runtime error will be produced. In our case, such an object is found—it is the one created on Line 1 and assigned to `emp:id1` on Line 2, so its method ‘rank’ is called.

Note that URIs can be assigned transitively (in our case the URI `emp::id1` could be assigned to another URI, such as `students:id7`). So an object could have several URIs assigned to it, and its methods could be called via any of the URIs. Our implementation covers the transitive URI assignment case, since the property `owl:sameAs` is transitive (and symmetric). When the `method_missing` method checks if RDF individual with URI `emp:id1` is the same as some other object, the logical inference is applied by a reasoner plugged into the Ruby-S environment. Applying inference ensures that the objects that are transitively (and also symmetrically) the same as the given RDF individual, are considered as candidates to respond to the method call.

Sharing properties

An interesting issue in URI assignment is that the URI, assigned to an object, in our case `emp:id1`, can actually designate an RDF individual with such URI. This individual can have its own properties (and methods, as will be explained below in Subsection 4.2.9). After the assignment is done, the properties of the RDF individual `emp:id1` can be accessed via the assigned object `p1`. For example, if the RDF individual `emp:id1` has a property `foaf:name`, this property could be accessed via the reference to the assigned object—`p1.foaf::name`. What we see here is that `emp:id1` and the object referenced by `p1` become virtually the same entity. Thus, we can access the properties and methods via both the URI and the reference. We saw how calling methods on the same objects/RDF individuals is implemented in the previous paragraph. Accessing properties of the same RDF individuals is accomplished transparently, using logical inference. An OWL reasoner, plugged into the Ruby-S environment, can infer that an RDF individual and a Ruby object have the same properties whenever they are linked by the `owl:sameAs` property.
Persisting Objects

The question arises what happens when a persistent object and a transient object are linked by a property. After the program that created the links terminates, the transient objects cease to exist and the links become invalid. Hence, the data representation of the persistent objects linked to the transient ones becomes invalid. A solution to this problem is to make persistent all the transient objects that are linked to persistent objects. This way all the objects linked to persistent objects will continue to exist between different runs of the program and no invalid links will be introduced into the data. Note that the solution deals with transitive links to the transient objects. Thus, once a transient object, linked to a persistent object, itself becomes persistent, all the transient objects linked to this object (either by properties or by references) must also become persistent. So, a transitive closure of all the objects linked from persistent objects must be calculated and all the objects in the closure must be made persistent. This solution also means that the garbage collector of the language must be changed to consider links from persistent objects (in addition to references from in-memory objects).

Implementation

For demonstration and code evaluation, we implemented only persisting of objects linked by the `owl:sameAs` property. While the work with the persistent objects is done directly on the data sources, the processing of transient objects is done in memory. Once the program terminates, the transient objects cease to exist. Due to the principle of Orthogonal Persistence described in Subsection 3.2.7, the objects must be persisted automatically, without requiring any involvement of the programmer. We implemented persisting the objects at the end of the program run and loading the persisted objects into memory during the consequent runs of the program.

Since our goal was not to change the Ruby language itself, we used the Ruby features `BEGIN` and `END`. These keywords of Ruby allow defining blocks of code that are executed by the Ruby execution environment before and after a Ruby program is executed, respectively. Our `END` block finds all the transient objects (the original Ruby objects) that are linked by the `owl:sameAs` property to some RDF individuals. The code finds the transient objects by a special URI pattern we defined for them. After finding the
objects, the code uses the `Marshal` module of Ruby to produce a byte stream representation of Ruby objects. This byte stream representation is stored in one of the data sources according to the “DNS” module. During the consequent program runs, the code in the `BEGIN` block reads the stored byte representations and reconstructs the objects from their stored byte stream representations.

This implementation ensures that the transient objects linked to persistent objects by the `owl:sameAs` property “survive” between different runs of the program, and this way become persistent.

Once an object is disconnected from a persistent object, i.e. the property `owl:sameAs` is removed, the object ceases to exist at the end of the program run. All this is done automatically and transparently by the Ruby-S execution environment, without interfering with the main code of the program and without requiring any involvement of the programmer, achieving Orthogonal Persistence. Transient objects are made automatically persistent by being linked to persistent objects and they are made transient again when the links to persistent objects are removed.

### 4.2.8 Orthogonal Persistence

The Orthogonal Persistence principle, explained in Subsection 3.2.7, means that the persistence issues must be handled by the language environment orthogonally to the main code of the program. In other words, the program code does not have to include explicit commands for loading or storing data, for making objects persistent or transient. This principle is implemented in Ruby-S by a combination of other implemented features. Since we enabled working with RDF individuals directly, by using regular operators such as dot, assignment and plus (Persistence Independence principle), no explicit loading/storing of RDF individuals in the code is required. We used “DNS” to specify data sources (Location and Migration Transparency principle) and assign URIs to objects (URIs and Access transparency principle), so a rule in “DNS” can specify that a transient object is made persistent, or migrated between data sources, without specifying this explicitly in the code. Because transient objects are persisted automatically once connected by properties to persistent objects (Property Attachment), no code is required for handling.

---

3http://ruby-doc.org/core-1.8.7/Marshal.html
the persistence issues that occur when transient objects are referenced by persistent objects. All these mentioned features enable handling persistent issues implicitly, behind the scenes. This way, the Orthogonal Persistence principle is upheld.

4.2.9 Procedure Attachment

The Procedure Attachment principle, explained in Subsection 3.2.5, states that all the language objects, including RDF individuals, can have methods. While Ruby objects naturally have methods, there is no such concept in RDF. Methods cannot be specified for RDF individuals and SPARQL queries cannot use methods. The only concept in SPARQL similar to methods is extension functions, however they can be applied only in the SPARQL FILTER and ORDER BY clauses. Hence, we had to create a syntax to specify methods of classes of RDF individuals, and to enable invocation of the methods in Ruby-S and in SPARQL queries.

Following the principle of Orthogonal Persistence, we implemented storing/loading methods to be transparent to the programmer. In our implementation, the programmer just defines a method in a class of RDF individuals, and the method is serialized and stored transparently. Later, the programmer can call the method, the method will be deserialized and will be executed by the language environment, without the involvement of the programmer. Following the URIs and Access Transparency principle, the methods will have URIs, and they will be accessible by URIs, regardless of where they are stored.

An additional principle that we pursued was to provide what we called property-method transparency. This means the ability to access properties and call methods using the same syntax. This way, it will be transparent to the programmer whether the result she gets is based on a value of a property or is a value returned by a method. Adding to the above the Integrated Logical Inference principle, it will be also transparent whether the result is inferred (or is stored in some data source or is calculated by a method). The following listing provides a code example for procedure attachment.

Listing 4.16: A procedure added to a class of RDF individuals

```
1 namespace 'http://www.acme.com/employees#', :emp
2 namespace 'http://www.acme.com/departments#', :dept
```
namespace 'http://www.acme.com/ontology#Department.Methods.',
  :deptmethods
namespace 'http://www.acme.com/ontology#', :ont

class ont::Department
  def totalSalary
    namespace 'http://www.acme.com/ontology', :ont
    deptID = ont::deptID
    (select q::salary where {
      q::employee ont::deptNo deptID;
      ont::salary q::salary
    }).inject(:+)
  end
end

print 'Total salary in department 1:'
print dept::id1.totalSalary

# calling a method by URI
print dept::id1.deptmethods::totalSalary

# accessing original and inferred properties
print dept::id1.ont::hasManager
print emp::id2.ont::manages

Listing 4.16 illustrates adding a method totalSalary to ont:Department class on Lines 8–17 and calling this method on an RDF individual of the class on Lines 20 and 23. Note that the regular Ruby class syntax is used for defining the class. (The name of the class is prefixed with the Ruby module name ont. We use here a Ruby module name to designate the URI namespace ont.)

The method is defined as a regular Ruby method. (Note the Ruby-S features we described before, such as embedding a SPARQL query inside the code of the method. The query returns the salaries of all the employees in the department the method is invoked on, and runs a loop to sum the
The method is called on Line 20 using the original Ruby syntax for calling methods. Line 23 illustrates calling the same method using the property access syntax of the Ruby-S—by a URI. Note that this URI is implicitly assigned to the method, it is of the form \texttt{<class name>\textunderscore Methods.<method name>}.

In our case, the method \texttt{totalSalary} defined for the class with URI: \texttt{http://www.acme.com/ontology\#Department} is automatically assigned the URI \texttt{http://www.acme.com/ontology\#Department\textunderscore Methods.totalSalary}.

Note that some other URI (an alias) can be explicitly assigned to any method, in a similar way to assigning additional URIs to Ruby objects described in Subsection 4.2.7. For example a statement \texttt{ont\textunderscore totalSalaryAlias} = \texttt{deptmethods\textunderscore totalSalary}, assigns an additional URI—\texttt{ont\textunderscore totalSalaryAlias}—
to the method \texttt{deptmethods\textunderscore totalSalary}. The method can be called by any URI which was assigned to it.

Note that once a method is defined in a class, it can be called in subsequent program runs since the method is serialized and stored transparently.

Lines 26 and 27 are shown to provide comparison with accessing actual and inferred properties of RDF individuals. On Line 26, the property \texttt{ont\textunderscore hasManager} which appears in one of the data sources, is accessed. On Line 27, the inferred property \texttt{ont\textunderscore manages} of the RDF individual \texttt{emp\textunderscore id2} is accessed (the Integrated Logical Inference is explained in Subsection 4.2.5 and shown in Listing 4.13). This exemplifies that calling methods by URIs, accessing actual and inferred properties of RDF individuals in Ruby-S is programmed using the same syntax, transparently to the programmer.

**Implementation**

First, we had to implement the following behavior—once a Ruby class is defined, its methods must be serialized and stored in one of the data sources. We had to add the “aspect” of serializing and storing methods to the regular class definition of Ruby. Here the challenge was to change class definitions in Ruby (without changing the Ruby itself).

We found the following solution—to use the \texttt{inherited} method in Ruby. This method can be defined in a class so that each time a subclass of that class is defined, the \texttt{inherited} method is called. The newly defined subclass is passed as a parameter. Once such method is defined in the \texttt{Object} class, it will be called for every new defined class, since every class in Ruby implic-
itly inherits from the *Object* class. We implemented the *inherited* method to inspect all the methods in the newly defined class and serialize them in RDF. We used Ruby reflection capabilities that allow querying classes for their methods. We used the *Ruby2Ruby* package[^1] for serializing/deserializing Ruby methods. This way we caused every method of every defined class to be serialized and stored in one of the data sources according to the “DNS” module.

Second, we had to implement executing of the serialized method once it is called on RDF individuals of the class. The problem here is that once a method is called on an RDF individual, for the Ruby execution environment the call appears to be performed on a temporary object that represents the RDF individual. For Ruby, this temporary object is not a member of the class for which the method is defined, so the Ruby execution environment cannot call the method directly on the temporary object. In the case presented in Listing 4.16, we can infer that RDF individual `dept::id1` belongs to the class `ont::Department` according to some ontology that resides in one of the data sources. But for Ruby, `dept::id1` is some temporary object and its relation to the class `ont::Department` is unknown. Hence, while the method `totalSalary` is defined for the Ruby class `ont::Department`, it cannot be called on the `dept::id1` temporary object by the regular Ruby method-call mechanism. We had to implement a special method-call mechanism for RDF individuals.

The method-call mechanism is implemented similarly to the property access mechanism described in Subsection 4.2.2, by using the `method_missing` method. (Recall that this method is invoked in Ruby when a non-existing method is called on an object. The name of the non-existing method is provided by the Ruby language environment as a parameter.) In our case, we changed the `method_missing` method to check whether a method with the name or with the URI of the called method, exists in one of the data sources. If such method is found in one of the data sources, its representation is loaded from the data source and deserialized. The deserialized method is added to the object on which the method is called (in Ruby, methods can be added to objects dynamically).

Note that the actual Ruby object involved here is some temporary object that represents the RDF individual (in our case `dept:id1`). An interesting

[^1]: [http://docs.seattlerb.org/ruby2ruby/](http://docs.seattlerb.org/ruby2ruby/)
issue is how to add the called method to that specific object, and not to all
the objects that represent RDF individuals. Here the following feature of
Ruby proves helpful—in Ruby, methods can be dynamically added not only
to classes, but also to specific objects. Such methods are called singleton
methods and are used to add behavior to specific objects instead of to all
the members of a class.

So, once the loaded and deserialized method is added to the object,
it is called and executed by a regular Ruby method-call mechanism. To
summarize, once a non-existing method is called, the Ruby-S environment
checks the data sources for the definition of this method, loads the stored
method representation, deserializes it, adds the method to the receiver object
and executes the method.

Note that here the Code on Demand paradigm [24] is implemented—
the method is brought from the data source (the server) to the execution
environment (the client) and is executed on the client. An alternative would
be to let the server (the data source) to execute the method and send the
client (the execution environment) the results—the Client-Server paradigm.
However, using Code on Demand is the right option, since in such case the
code is executed in the context of the Ruby-S execution environment. While
the context of a single data source includes only the data in that source, the
context of the execution environment includes the data in all the federated
data sources, and also the data inferred by reasoning. Hence, all the methods
in Ruby-S must be executed in the context of all the federated data sources
while considering the inferred data, without any regard to where the method
is stored (Location Transparency).

**Calling a method from a SPARQL query**

Listing 4.17 shows how the totalSalary method we defined previously can be
called from a SPARQL query. Note that this is the original SPARQL syntax
that is executed by a SPARQL query engine. It is not the syntax of embed-
dded SPARQL queries in Ruby-S, as appears, for example, in Listing 4.6.
This feature shows that we not only brought Semantic Web features (such
as SPARQL queries) to Ruby, but also brought Ruby features (methods) to
the Semantic Web (SPARQL).

Listing 4.17: Calling a method from a SPARQL query
In the SPARQL query in Listing 4.17, the call to the method `totalSalary` appears in the triple pattern of the query. The call to the method appears as a regular RDF property and the query follows the standard SPARQL syntax. The result returned by the method is placed into the query variable `?total`. This way, the same syntax is used for variables calculated both based on the actual values of RDF properties and based on execution of methods. Here we have an interesting situation that shows the ability to mix Ruby language constructs and Semantic Web constructs in Ruby-S. We call a method defined on a class of RDF individuals inside a SPARQL query, while the method uses another SPARQL query (shown in Listing 4.16) for its calculations. The SPARQL query in Listing 4.17 can, in turn, itself be embedded in Ruby-S.

**Implementation**

Implementing the ability to call a Ruby method from a SPARQL query is related to SPARQL, and not directly to Ruby-S. Hence, for this feature we had to change a SPARQL query engine instead of extending the Ruby programming language, as we did when we implemented other features.

We extended the ARQ query engine [1], version 2.8.0, that we used for our execution environment, to perform calling methods from SPARQL queries. We used the property functions mechanism of ARQ. In ARQ special functions can be defined to appear in SPARQL queries as if they are RDF properties. These functions can be dynamically loaded or pre-registered, and they are invoked during the query execution.

For calling methods of Ruby-S from SPARQL queries executed by ARQ, we used a “fake” URI schema—“rosw://” (rosw for Ruby-S). We registered a handler in ARQ to be called once a property with the schema “rosw://” is encountered. We used the same mechanism used in ARQ for dynamically load-
ing property functions implemented in Java by using the fake URI schema “java://”. Once called, our handler reads the postfix of the “property”, e.g. totalSalary, and calls the Ruby-S execution environment to execute the method on the RDF individual of the property. The result returned by the Ruby-S execution environment is fed back into the ARQ query engine which continues evaluating the query based on the returned result. The returned result here is treated by the evaluation of the query as if it represents a value of an RDF property.

4.2.10 Multiple Dynamic Class Membership

The Multiple Dynamic Class Membership principle, explained in Subsection 3.2.6, states that any object can be a direct member of multiple classes simultaneously, similarly to class membership in the Semantic Web. We repeat here the definition of direct class membership provided in Subsection 3.2.6: An object \( o \) is a direct member of a class \( C \), if \( o \) is a member of \( C \) and \( o \) is not a member of any class that inherits from \( C \).

To explain the involved issues with the implementation of this principle, we provide below background on class membership in Ruby and in the Semantic Web (in RDFS and OWL).

Background on the Meaning of Classes in Ruby and OO

In Ruby, all the methods of all the classes of an object can be invoked on the object. Methods in subclasses can override the methods defined in the superclasses, according to the inheritance hierarchy.

Additional meaning of the class membership is that an object can be queried if it belongs to a (direct or indirect) class. The direct class membership of an object is declared once the object is created. For example, the following statement creates an object of the class Student: \( s = \text{Student.new(“John”, “Doe”)}. \) The indirect classes of the object are determined according to the class hierarchy. The class hierarchy is determined by inheritance relationships between classes, and is created during the definition of the classes. Continuing the previous example, if the Ruby class Student is defined to inherit from the class Person, the newly created object will have the class Person among its indirect classes. (It will also have the class Object as one of its indirect classes, since all the classes in Ruby implicitly
A unique feature of Ruby is that every object is a member of two direct classes—the class of the object and the eigenclass of the object. The class of the object holds all the methods shared by all the instances of the class. The eigenclass is an anonymous class, implicitly attached to the object. It holds methods defined on the object. The methods defined on the object are not shared with other objects. The method lookup is performed by the Ruby execution environment first on the eigenclass and then on the direct and indirect classes of the object, according to the inheritance hierarchy, from subclass to superclass. There is no multiple inheritance in Ruby. To summarize, in Ruby the class membership of an object is manifested by the methods that can be invoked on the object and by querying the object to find the classes it belongs to.

An additional point to mention in the OO terminology, is separation of a message an object receives, from the method that the object performs as a result of receiving the message. For example, in Java, we can have the class Employee which extends the class Person. Each of these classes can have a method with the same name, for example foo. Note that a reference of the declared type Person can reference an object of the class Employee since the class Employee extends the class Person.

Listing 4.18 illustrates the method dispatch in OO languages using an example in Java.

Listing 4.18: Example of method dispatch in Java

```
1 class Person {
2     String foo() { return "foo of Person"; }
3     String bar() { return "bar of Person"; }
4 }
5 ...
6 class Employee extends Person {
7     String foo() { return "foo of Employee"; }
8 }
9 ...
10 Person p1 = new Person();
11 Person p2 = new Employee();
12 p1.foo(); // "foo of Person" is returned
```
On Lines 1–4 and on Lines 6–8 of Listing 4.18, the classes Person and Employee are created. (In Java the classes have to be defined in separate files, however here we show the classes in one listing to save space.) The classes have methods Person.foo(), Person.bar() and Employee.foo(). On Lines 10 and 11, objects of the classes Person and Employee are created and assigned to references of the type Person. On Lines 12 and 13, the message “foo” is sent to both objects. This message causes two different methods to be executed. When the message “foo” is sent to an object of the class Person, via the reference p1, the method Person.foo() is executed. However, when the message “foo” is sent to an object of the class Employee via the reference p2, the method Employee.foo() is executed, rather than the method foo() of the super class—Person.foo(). This is because the method foo is redefined in the class Employee. It is said that the method foo of the class Employee overrides the method foo of the class Person. The fact that the message is sent via a reference of type Person does not change the method dispatch—the method to be executed is chosen dynamically according to the actual type of the referenced object.

On Lines 14 and 15, the message “bar” is sent to the objects created on Lines 10 and 11. As a result, the same method—Person.bar() is executed on Lines 14 and 15, since this method is not redefined in the class Employee.

We would like to stress the difference between the message sent to an object and the method that is executed as a result. In Java and in many other OO languages, the message is exactly the same as the name of the method.

**Background on the Meaning of Classes in the Semantic Web**

While in Ruby the classes of the objects are declared, in the Semantic Web, the classes of RDF individuals are either declared or implied. While in Ruby the classes of an object define which methods can be invoked on the object, in the Semantic Web the relation between classes and properties of an RDF individual is inverse—the properties of an RDF individual imply its class. Any property can be “attached” to any RDF individual, in other
words, any RDF individual can appear as the subject of any property. Each
property can determine a direct class of its subject—this is done by stating
an ontological axiom using the property rdfs:domain. For example, it can
be stated in some ontology that the domains of the properties ont:hasSalary
and ont:studiesAt are the classes ont:Employee and ont:Student, respectively.
The properties ont:hasSalary and ont:studiesAt can be attached to any RDF
individual, and once they are attached to an individual, it is implied that its
direct classes are ont:Employee and ont:Student. In addition to the implied
classes, it can be declared that an RDF individual belongs to other classes
by using the rdf:type property.

Similarly to the inheritance (subclass-superclass) relationship between
classes in OO, there is a subclass-superclass relationship between classes in
the Semantic Web. It can be stated in OWL that the class ont:Employee is
owl:subClassOf the class ont:Person. The superclass of all the RDF classes
is the class owl:Thing and as such it is the class of all the RDF individuals.
The concept of a superclass of all the classes is similar to the class Object
in Ruby and the class java.lang.Object in Java. (There is also the class
owl:Nothing which is a subclass of all the RDF classes).

Every RDF individual is a member of the implied classes, of the declared
classes, and of all the superclasses of the implied and the declared classes.
For example, suppose the classes ont:Employee, ont:Student, ont:ChessPlayer
have the class ont:Person as their superclass. If an RDF individual ont:John
has properties ont:hasSalary and ont:Student, and also the property rdf:type
with the value ont:ChessPlayer, it belongs to the following classes:

- Implied classes ont:Employee and ont:Student
- Explicitly declared class ont:ChessPlayer
- The superclass of the implied and the declared classes—ont:Person
- The owl:Thing class which is the superclass of all the classes

The classes in the Semantic Web serve mostly a descriptive, and not
a prescriptive purpose. The classes do not determine which properties an
individual must have. On the contrary, the properties of an RDF individual
determine which classes the individual belongs to.

Another use of classes in the Semantic Web is validation. One of the
possible ontological axioms is stating that two classes are disjoint, i.e. no
individual can belong to both classes. For example, it can be declared that class Male and class Female are disjoint. Suppose for example, that some data source contains the triples: (ont:Alex ont:isBrotherOf ont:Bob) and (ont:Alex ont:isSisterOf ont:John). Suppose also that some ontology states that the domains of the properties ont:isBrotherOf and ont:isSisterOf are classes ont:Male and ont:Female, respectively. In addition, the ontology states that these classes are disjoint. A reasoner can infer that ont:Alex belongs to both the ont:Male and the ont:Female classes and report about the inconsistency.

**Multiple Class Membership in Ruby-S**

Since in SWOOM, RDF classes have methods, the method dispatch must be changed to reflect the RDF class membership. So the method dispatch must be performed according to all the classes of an RDF individual.

In Ruby-S we introduce separation between the URI of the method and between the message, to which the method is executed as a response. We introduce different method call semantics—one by the message and another one by the URI of the method. When the method is called by the URI, this exact method is executed disregarding the type of the receiving object. When a method call by message is executed, method dispatch is performed—the method must be found in the classes of the object in order to be executed as a result of the call.

The issue with the method dispatch in Ruby-S is how to handle a situation where multiple classes of the object have methods that can be executed as a response to the message. Note that in Ruby-S an object can belong to multiple direct and indirect classes. Which method should be chosen among multiple candidate methods? We believe that the natural semantics for Ruby-S is the semantics of the OO languages with multiple inheritance, such as the semantics of C++ [57].

In C++, the chosen method is the method that overrides all other candidate methods (the method in the most specific class, i.e. the class that is a subclass of all the classes of all the other candidate methods). If no candidate method is found, an error is reported by the compiler. Alternatively, if there exist several candidate methods in multiple indirect classes of the object and none of them is the most specific method, an error of a different kind is returned to designate this situation.
So, we extend the semantics of C++ to be applied to Ruby-S with the only change that the candidate methods in Ruby-S appear both in the direct classes of the object and in the indirect classes of the object. (In C++ an object is a member of one direct class and can be a member of multiple indirect classes.) If no candidate method is found in both the direct and the indirect classes of the object, an error is returned, during run time. (Note that there is no possibility to perform the check for candidate methods during compile time since methods can be added or removed during run time.) If no method that overrides all other candidate methods is found, an ambiguity error is returned, during run time. (Here, again, no compile time checks can be performed.)

Listing 4.19 exemplifies the method dispatch in Ruby-S.

Listing 4.19: Multiple class membership and method dispatch

```ruby
namespace 'http://www.acme.org/ontology#', :ont
namespace 'http://www.acme.com/ontology#Student::Methods.',
  :studentmethods

class ont::Person
  def printTitle; puts 'I am a Person'; end
end

class ont::Employee
  def printTitle; puts 'I am an Employee'; end
end

class ont::Manager
  def printTitle; puts 'I am a Manager'; end
end

class ont::Student
  def printTitle; puts 'I am a Student'; end
end

# Explicitly specify the type of ont::john
```

81
ont::john.rdf::type+= [ ont::Person ]
# now John is a Person

ont::john.printTitle "'I am a Person' is printed"

ont::john.ont::salary = 100
# now John is an Employee

ont::john.printTitle "'I am an Employee' is printed"

ont::john.ont::isManagerOf = [ ont::Bob ]
# now John is a Manager

ont::john.printTitle "'I am a Manager' is printed"

ont::john.ont::studiesAt = ['Technion']
# now John is both a Manager and a Student

ont::john.printTitle "Ambiguity Exception is thrown"

# calling printTitle method of Student by URI
# 'I am a Student' is printed
ont::john.studentmethods::printTitle

ont::john.ont::salary = nil
ont::john.ont::isManagerOf = []
# now John is a Student

ont::john.printTitle "'I am a Student' is printed"

ont::john.ont::studiesAt = []
# now John is only a Person

ont::john.printTitle "'I am a Person' is printed"
Listing 4.20 shows an ontology for Listing 4.19.

Listing 4.20: An ontology for Listing 4.19

```r
@prefix rdfs:<http://www.w3.org/2000/01/rdf-schema#>.
@prefix ont: <http://www.acme.org/ontology#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

ont:Employee
    rdfs:subClassOf ont:Person .

ont:salary
    a owl:FunctionalProperty ;
    rdfs:domain ont:Employee .

ont:Manager
    rdfs:subClassOf ont:Employee .

ont:studiesAt
    rdfs:domain ont:Student .

ont:isManagerOf
    rdfs:domain ont:Manager ;
    rdfs:range ont:Employee .

ont:Student
    rdfs:subClassOf ont:Person .
```

The ontology in Listing 4.20 specifies that the class *Employee* is a subclass of *Person*, the class *Manager* is a subclass of *Employee* and the class *Student* is a subclass of *Person*. The domains of the properties *ont:salary*, *ont:isManagerOf* and *studiesAt* are classes *Employee*, *Manager* and *Student*, respectively. In other words, once these properties are set for some individual, this individual becomes a member of the respective classes.

Listing 4.19 defines the method *printTitle* for all the involved classes on Lines 6–20. On the following lines, the properties specified in the ontology
are set and unset for the individual ont:john, effectively changing the classes of the individual. The method printTitle is called on ont:john after every change in the class membership of the individual.

Each time a message is sent to an individual, the method in the most specific class of the individual is called. For example, when ont:john is Employee, on Line 31, the printTitle method of the Employee class is called. Once the property ont:isManagerOf is set for ont:john, on Line 33, it becomes a member of the class Manager. The class Manager is a subclass of the class Employee, so it becomes a more specific class of ont:john. On the following line, the method printTitle of the class Manager is called.

An interesting case is shown on Lines 38–41. On Line 38, the property ont:studiesAt is set for ont:john. Once the property is set, ont:john becomes a member of the class Student, in addition to being a member of the class Manager. Since Student and Manager are not subclasses of one another, no most-specific class of ont:john exists in this case. We have a situation where the method printTitle is defined in two classes—Student and Manager—and none of them is more specific than the other, so an ambiguity exception is thrown by the execution environment.

Line 45 shows calling the method printTitle of the class Student, by URI. Note that here we distinguish between calling the methods by “message”, such as printTitle, and by URI, such as studentmethods::printTitle. In the former case, the method dispatch is done according to the class hierarchy, i.e. the method in the most specific class is called. In the latter case, the class membership of the individual is not taken into account, i.e. the method specified by the URI is the method that is called.

**Implementation**

In implementing the call by message described above, we could not use the regular method dispatch mechanism of Ruby, since the interpreter of Ruby is not aware of the RDF class hierarchy, and as we stated previously we did not want to change the interpreter of Ruby. We implemented call by message by implementing calling serialized methods, as explained in Subsection 4.2.9. Once a method for an RDF class is defined, its code is serialized and it is automatically assigned a URI. The message by which the method is defined, e.g. printTitle, is stored in the data source beside the serialized code. Later, the method can be called either by its message or by its URI.
When a method is called by a message, all the multiple classes of the individual are queried to check which classes contain a method with such message. If no such class is found, a NoMethodError exception is thrown. If several classes that contain such method are found, they are queried to find the most specific class. If no most specific class is found (there are at least two classes that contain a method with the message, and neither of the classes is a subclasses of the other), an AmbiguousMethodCall exception is thrown. If the most specific class is found, the method of the class is deserialized and executed as described in Subsection 4.2.9.

When a method is called by URI, the method designated by the URI is deserialized and called, disregarding the class hierarchy.

4.2.11 Transaction Management

As explained in Subsection 3.2.12, in order to enable correct updates to the persistent data by programs executed concurrently by the language environment and to provide atomicity of updates to the persistent data, transactions should be part of the language.

The Semantic Web query and update languages, SPARQL and SPARQL/Update do not yet support transaction management, so we did not add transaction management functionality to Ruby-S. The transaction management can be performed in Ruby-S in a similar way to the way it is performed today in federated databases. The data sources must support an Atomic Commit protocol, e.g. Two-Phase Commit protocol (2PC) (the protocol is described by Bernstein et al. [17]). The Ruby-S language environment must serve the role of the coordinator in the protocol while the data sources serve the role of the participants.

When the data sources apply pessimistic locking, the Two-Phase Commit protocol guarantees Global Serializability. Global Serializability is a property of the unified schedule of all the transactions in all the federated data sources. Global Serializability means that the unified schedule is serializable and the local schedules of all the data sources are serializable as well.

For data sources to have optimistic (non-blocking) transactions, the Commitment Ordering [52] technique must be applied, in order to provide Global Serializability. If some data sources do not support Atomic Commit protocol, Global Serializability could not be provided. Such data sources
may still provide local serializability, i.e. the applications that process the data locally will operate in a serializable way.

Read-only data sources do not have to support transactions or an atomic commit protocol. The read-write data sources that do not support transactions, could be locked as a whole, preventing concurrent access. In case of an abort of a distributed transaction, the Ruby-S execution environment should be able to undo the changes made since the beginning of the transaction, so it should implement logging of operations for non-transactional data sources.

4.2.12 Model Transparency

The Model Transparency principle, explained in Subsection 3.2.8, states that a program in SWOOM must be written as if all the data being processed are in RDF, while actual data are represented in different models and stored in different formats.

This principle is supported in the Ruby-S environment by federating heterogeneous data sources via adapters to RDF, as explained in Section 4.1. One of the problems with existing adapters to RDF is that currently most of them are read-only. In other words, most of the adapters today support only SPARQL queries without supporting SPARQL/Update statements. For example, we used the popular relational-to-RDF mapping framework, D2RQ, for connecting relational databases to the Ruby-S environment. D2RQ, however, supports only SPARQL queries. In order to enable updating relational data mapped to RDF, we had to develop our own extension\(^5\) to the D2RQ framework, that supports SPARQL/Update statements [29]. Since this extension is not in the focus of this thesis, we do not provide the details here.

The processing in Ruby-S of all the data items (including the original Ruby objects) is performed by using SPARQL queries, reading and updating RDF properties, independently of and obliviously to the original model of the data items, be it Relational, OO or any other model.

\(^5\)http://d2rqupdate.cs.technion.ac.il/
Chapter 5

Optimizations

One of the benefits of SWOOM is that the compiler or the interpreter of a SWOOM programming language can consider Semantic Web artifacts during its optimizations. Optimization opportunities stem from the unified view of the compiler on programming language constructs, such as loops and arithmetic operations, and on Semantic Web constructs such as queries and RDF properties. An interesting opportunity stems from using reasoning in optimization decisions.

For comparison, consider a compiler of a regular OO language. When an application in a regular OO language performs SPARQL queries by sending a query string to some query engine, the query string is opaque to the compiler. That is, the compiler does not parse the query and cannot use any information about the query for its optimizations.

Next, we describe several optimizations that we propose for SWOOM.

5.1 Aggregation Pushing

In this section we describe the optimization we called Aggregation Pushing. We implemented this optimization in Ruby-S and in the ARQ\(^1\) execution engine, and measured the provided improvement.

To perform the optimization, the compiler “pushes” aggregation computations, such as summations, from loops that process results of SPARQL

\(^1\)http://jena.apache.org/documentation/query/
queries\(^2\), into the queries themselves. Once the aggregation is moved into the query, the query can be executed entirely on the site of the data source and only the aggregated result will be sent back to the Ruby-S environment, instead of sending the data itself. This decreases network bandwidth, and the computation can be performed directly in the data source. For example, in cases where the underlying data source is an RDBMS, mapped to RDF, the aggregated SPARQL query can be further translated to an SQL query with aggregation. If the RDBMS is capable of optimizing the aggregation, even more performance improvement will ensue.

For example, consider the code in Listing 5.1. We ran this code on the Berlin SPARQL benchmark [18]. The code calculates the sum of all the ratings in the `review` table of the benchmark.

**Listing 5.1: Summation on the results of a query**

```ruby
sum = (select q::rating where {
    q::review bsbm::rating1 q::rating
}).inject(:+)
```

The code in Listing 5.1 performs a SPARQL query and executes the `inject` method of Ruby on the results of the query. The `inject` method performs aggregation on the results of the query. In the case of Listing 5.1, it receives symbol `+` as a parameter, so summation is performed.

We implemented Aggregation Pushing by transforming the code in Listing 5.1 to the SPARQL query shown in Listing 5.2. Note that the summation is moved from the `inject` method into the query.

**Listing 5.2: Optimized SPARQL query for the code shown in Listing 5.1**

```sparql
SELECT SUM(?rating) WHERE {
    ?review bsbm:rating1 ?rating .
}
```

For cases when the queried data is mapped from a relational database to RDF, we optimized the resulting SPARQL query even further by pushing the aggregation into the SQL query to be executed on the relational data source. The D2R relational-to-RDF mapping framework [19] does not perform this

---

\(^2\)The SPARQL 1.1 Working Draft [37] introduces aggregation functions, such as COUNT, SUM, MIN, MAX, AVG, into the query language.
optimization, so we implemented this optimization for the D2R framework by ourselves. The resulting SQL query is shown in Listing 5.3.

Listing 5.3: Optimized SQL query for the code shown in Listing 5.1

```
SELECT SUM(rating1) FROM review
```

5.1.1 Implementation

We implemented the optimization in Ruby-S while maintaining our strategy of not changing the Ruby interpreter. We wrote a code that translates the SPARQL query in Listing 5.1 and inserts the summation into the resulting query. The problem here is that the summation (the method `inject`) is applied to the results of the query. In a straightforward implementation, the query must be executed before the method `inject` is called. Our implementation, described in Subsection 4.2.3, cannot “look ahead” to know that the method `inject` will be executed on the results of the query.

Our solution was to implement the optimization by using the Virtual Proxy Design Pattern [35]. Using this pattern, lazy query evaluation is performed, i.e. instead of executing the query and returning the results, a proxy object is returned. This proxy object “remembers” the query to be executed. Once the method `inject` is called on this proxy, the proxy inserts the aggregation into the query, sends the query for execution, and returns the results. If some method other than `inject` is called on the proxy, the proxy just sends the query for execution without any change and returns the results.

To implement this pattern we changed the `select` method to return the proxy instead of executing the query. We implemented the `inject` method of the proxy to push the aggregation into the query before sending the query for execution. For all other methods called on the proxy object, we added an “aspect” of executing the query and performing the methods on the results of the query. Such an aspect can be added in Ruby to all the methods of a class by querying the methods of the class, creating copies of the methods by using the Ruby `alias` keyword and by creating new versions of the methods. These new versions execute the query and call the original versions of the methods on the results of the query.

To implement the optimization of pushing the aggregation from SPARQL
to SQL, we extended the query engine of D2R. We registered an object of a class we created—*QueryEngine*, which extends the *QueryEngineD2RQ* class of D2R, in the query engine registry of the ARQ query processor that we used in our system. The ARQ query processor allows query engines to be plugged into its registry (*QueryEngineRegistry* class). After parsing a SPARQL query, the ARQ query processor finds a query engine capable of handling the query. For this purpose, the query processor polls the registered query engines if they match the queried RDF graph. Among the matching query engines, the last one to register is selected to handle the query. This way new query engines can be plugged into the ARQ query processor and can override its query execution.

In our query engine we extend the operation of the query engine of D2R to consider aggregation operations of SPARQL queries. Our query engine adds the aggregation operations to the corresponding SQL queries created and executed by the original D2R query engine.

### 5.1.2 Experimental Evaluation

The code in Listing 5.1 executes the query and the aggregating loop without any optimization. Listing 5.2 shows the aggregation pushed into the SPARQL query. Listing 5.3 shows the same aggregation pushed into the corresponding SQL query. We compared the performance of all the three cases.

We ran the code on the table *review* of the Berlin SPARQL Benchmark [18]. The table contains 1.2 million rows. We performed the measurements on a computer with Intel(R) Xeon(R) E5540 processor, 2.53 GHz, memory size 4 GB, cache size 8 MB, running the Linux Red Hat 4.1.2 operating system. We used PostgreSQL 8.1 as the database management system, D2RQ 0.7 to map the relational data to RDF, Joseki SPARQL server 3.4.0, ARQ SPARQL query processor 2.8.0, and Jena 2.6.0 for RDF representation.

The results we measured were as follows: pushing the aggregation from Ruby-S into the SPARQL query reduced running time by a factor of 40. Such drastic improvement can be attributed to the slow execution of Ruby, in general, since it is an interpreted language. The aggregation performed by the ARQ query processor, written in Java, was much faster than the Ruby
Figure 5.1: The impact of pushing aggregation into SPARQL and SQL. The labels on the X axis specify where the main computation was performed, Ruby-S or SPARQL. The words in parentheses specify where the aggregation was performed, Ruby-S, SPARQL or SQL. The Y axis presents the running time, in centiseconds, on a logarithmic scale.

Pushing the aggregation from the SPARQL query into the SQL query improved the performance further by a factor of 12. This performance improvement stems from the fact that this time the aggregation was performed by the PostgreSQL DBMS (written in C), close to the physical data, without the need to transfer all the aggregated data items from the database to the ARQ query processor.

Figure 5.1 shows the measurements we performed for the code shown in Listing 5.1. It uses logarithmic scale on the Y axis, in tenths of seconds.
The X axis contains labels that designate the original and the optimized calculations. The labels designate where the main calculation is performed, either in Ruby-S or directly via a SPARQL server. The word in parentheses designates where the aggregation is performed: in Ruby-S, in the SPARQL query engine or in the SQL query engine. So, the original code, where both the main calculation and the aggregation are performed in Ruby-S, is designated by the label *Ruby-S (Ruby-S)*. The optimized code, executed in Ruby-S, with aggregation performed in the SPARQL query engine, is designated by the label *Ruby-S (SPARQL)*. The optimized code, executed in Ruby-S, with aggregation pushed further to the SQL query engine, is designated by the label *Ruby-S (SQL)*.

We also measured the impact of pushing aggregation from SPARQL to SQL without regard to Ruby-S, by running the SPARQL query of Listing 5.2 directly on the SPARQL server. *SPARQL (SPARQL)* and *SPARQL (SQL)* designate executions of the SPARQL query directly on the SPARQL server. The *SPARQL (SPARQL)* label refers to a run of the query on the SPARQL server, with the aggregation performed in the SPARQL query engine. The *SPARQL (SQL)* label refers to running the query on the SPARQL server, with the aggregation performed in the SQL query engine.

While pushing the aggregation from Ruby-S to SPARQL showed improvement by a factor of 40, pushing the aggregation from SPARQL to SQL showed improvement by a factor of 12, both when the query was invoked from Ruby-S and directly on the SPARQL server.

### 5.2 Condition Pushing

In this section we describe the optimization we called *Condition Pushing*. This optimization moves condition statements from the programming language into SPARQL queries.

For example, consider the code in Listing 5.4.

Listing 5.4: Condition inside an iteration over query results

```ruby
1
2 (select q::emp where {
3   q::emp rdf::type ont::Employee
4 }).each { |emp| |
```

92
The code iterates over the results of a SPARQL query that returns all the RDF individuals of the type `ont:Employee`. Inside the loop, the department number of an employee is compared to 7 as a condition to further processing. To perform the optimization, the compiler pushes the comparison of the department number into the query, producing the code equivalent to the code shown in Listing 5.5.

Listing 5.5: Optimized version of the code shown in Listing 5.4

```ruby
if (emp.ont::deptNo == 7) {
    # do some processing
    ...
}
```

The query in Listing 5.5 selects the employees of Department 7 locally in the data source, eliminating the need to bring all the data items from the source into the SWOOM execution environment. The filtering itself can be performed more effectively in the data source, especially if the data items are indexed by the department number.

### 5.2.1 Experimental Evaluation

We evaluated the potential of the optimization by performing it manually in the code. We ran the original and the modified versions of the loop on the Berlin SPARQL Benchmark [18], using the setting described in Subsection 5.1.2. We ran the code in Listing 5.6, the listing shows two loops we compared. The first loop contains the conditional in the body of the loop, in other words it is the non-optimized version of the loop. The second loop contains the conditional as part of the query, thus it simulates the optimized version of the loop.
Listing 5.6: The simulated manually Conditional Push optimization

```ruby
# the non-optimized loop
(select q::rating where {
  q::review bsm::rating1 q::rating
}).each { |rating|
  if (rating == 2) then
    count+= 1
  end
}

# the optimized loop
(select q::review where {
  q::review bsm::rating1 2
}).each { |review| count+= 1 }
```

The code in both loops selects ratings with value 2 out of the `review` table of the Berlin SPARQL Benchmark. Rows with this rating constitute 7% of all the rows in the table. The non-optimized version of the code brings ratings of all the rows into the federation site and selects the ratings with value 2 in the body of the loop. The optimized version of the code performs the selection inside the query, so only the selected ratings are transferred from the SPARQL endpoint to the federation site. It saves the network bandwidth by eliminating the need to transfer the values of all the rows. We measured performance improvement by a factor of 9 in the simulated optimized version.

### 5.3 Join Pushing

In this section we describe the optimization we called Join Pushing. The optimization pushes “joins” performed in the programming language into SPARQL queries.

To perform the optimization, the compiler detects situations in which a join of two sets of data items is performed “procedurally”. For example, the join can be performed by iterating over the results of two queries and comparing the values of the shared properties, as shown in Listing 5.7.
Listing 5.7: “Procedural” join on the results of two SPARQL queries

```sparql
(select q::emp q::deptNumber where {
  q::emp ont::deptNumber q::deptNumber}).each {
  |emp, deptNumber|

(select where {
  q::dept ont::deptNo deptNumber}).each {
  |dept|
  # perform some processing on
  # the department and the employee
  ...
}
}
```

The first query in Listing 5.7 returns all the RDF individuals representing employees in the data sources, together with their department numbers. For each employee, another query is executed, which returns all the RDF individuals representing departments in the data sources, with the same department number as the department number of the employee. Note that there are two loops in the code, one nested in the other.

The compiler can push the “procedural” join into a SPARQL query, producing the code in Listing 5.8, equivalent to the code shown in Listing 5.7.

Listing 5.8: Optimized version of the code shown in Listing 5.7

```sparql
(select q::emp q::dept where {
  q::dept ont::deptNo q::number .
  q::emp ont::deptNumber q::number.
}).each {
  |emp, dept|
  # perform some processing on
  # the department and the employee
  ...
}
```

Performing the join on the site of the data source can be executed faster on the source, than procedurally on the site of the execution environment. The difference will be even more in favor of the data source, when the shared attributes are indexed in the source.
5.3.1 Experimental Evaluation

We evaluated the potential of the optimization by performing it manually in the code. We ran the original and the modified versions of the code on the Berlin SPARQL Benchmark [18], using the setting described in Subsection 5.1.2. We used a small version (50K) of the benchmark, since on larger versions computing the joins was too slow. We ran a join on the product attribute of the review and offer tables of the benchmark. Each table contains about 2000 rows. We created indexes on the joined attribute for both tables.

We ran the code in Listing 5.9 (the listing shows snippets we compared). The first snippet is a join performed by two loops. It is the non-optimized version of the code. The second snippet performs the join inside the query, thus it simulates the optimized version of the code.

Listing 5.9: The manually simulated Join Push optimization

```ruby
# the non-optimized version of the code
(select q::review q::rating where {
    q::review bsbm::rating1 q::rating
}).each { |review, rating|
    productno = review.bsbm::productno[0]
    (select q::offer where {
        q::offer bsbm::productNumber productno
    }).each { |offer| count+=1 }
}

# the optimized version of the code
(select q::review q::rating q::offer where {
    q::review bsbm::rating1 q::rating;
    bsbm::productno q::productno .
    q::offer bsbm::productName q::productno
}).each { |review, rating, offer| count += 1 }
```

The code in both snippets selects reviews and offers joined on their product number. The non-optimized version of the code selects all the tuples from the table review. For each review, the code performs another query to
return offers with the same product number.

The optimized version performs the join on the data source. It reduces the network bandwidth since only the joined rows are returned. In addition, the join is performed by an RDBMS, which is optimized for performing such operations. The indexes on the joined attributes should also contribute to the performance of the join performed by the RDBMS. The optimized version eliminates the need to execute a large number of queries in a loop by performing the join by a single query.

We measured performance improvement by a factor of 2 in the simulated optimized version.

5.4 Bulk Update

In this section we describe the optimization we called Bulk Update. This optimization moves multiple update statements performed in a loop, into a single update statement performed in bulk.

For example, consider the code in Listing 5.10.

Listing 5.10: Update of a property inside a loop

```ruby
1 (select q::emp where {
2   q::emp emp::deptNo 7}).each { |emp| 
3   emp.ont::salary = 1000;
4 }
```

The code in Listing 5.10 iterates over the results of a SPARQL query that returns all employees in Department 7, as RDF individuals. Inside the loop, the salary of each employee is set to 1000. The compiler can push the update of the property `ont::salary` into the query, producing the code equivalent to the code shown in Listing 5.11.

Listing 5.11: Optimized version of the code shown in Listing 5.10

```ruby
1 insert { q::emp ont::salary 1000 } where {
2   q::emp ont::deptNo 7 }
```

The code in Listing 5.11 shows SPARQL/Update query embedded in Ruby-S. The equivalent query in the original SPARQL/Update syntax is shown in Listing 5.12.
Performing the update directly in the data source is faster. It reduces the transfer of the query results and of the update statements between the program and the DBMS server.

5.4.1 Experimental Evaluation

As with the two previous optimizations, we evaluated the potential of this optimization by performing it manually in the code. For this purpose, we compared update of multiple rows performed in a loop with a SPARQL/Update statement performed in plain Ruby. We ran the original and the modified versions of the code on the Berlin SPARQL Benchmark [18], using the setting described in Subsection 5.1.2. We used a small version (50K) of the benchmark, since for larger versions the update took too much time. We ran the update on about 7% of the rows of the review table of the benchmark. This table contains about 2000 rows.

Listing 5.13 shows the code snippets we compared. The first snippet contains an update performed in a loop. The second snippet shows the update performed by a SPARQL/Update statement.

```
Listing 5.13: The simulated manually Bulk Update optimization
1 # the non-optimized version of the code
2 (select q::review where {
3   q::review bsbm::rating1 2
4 }).each { |review| review.bsm::rating1 = [12] }
5
6 # the optimized version of the code, in plain Ruby
7 ::Net::HTTP.post_form(URI.parse(targetURI),
8   {'request',
9     'PREFIX bsm: <http://www4.wiwiss.fu-berlin.de' +
10      '/bizer/bsbm/v01/vocabulary/>\n' +
11     'MODIFY DELETE { ?review bsm:rating1 2 }\n' +
12     'INSERT { ?review bsm:rating1 12 }\n' +
13     'WHERE { ?review bsm:rating1 2 }\n'})
```
The code in both snippets replaces all ratings of 2 by ratings of 12, for tuples in the table `review`. The non-optimized version of the code selects all the reviews with rating 2 from `review`. For each tuple, the code sets the rating to be 12. The optimized version of the code performs the update inside the SPARQL/Update statement.

We measured performance improvement by a factor of 27. We see such a drastic performance improvement here due to the fact that in the optimized version no results were transferred to the federation site and the whole update was performed as one statement. Compare with the non-optimized version, where the tuples to be updated were transferred to the federation site, and the update statements were sent to the data source one by one, a statement per updated data item.

5.5 Query Prefetching and Results Caching

In this section we describe the optimization we called Query Prefetching and Results Caching. When a value of a property of an individual is accessed, the SWOOM execution environment can optimistically bring values of other properties of the individual, avoiding a number of round-trips between the data source and the SWOOM execution environment. In a case when the data of the individual reside in an RDBMS mapped to RDF, and when the values of all the properties appear in one row of one table, such optimization can be especially beneficial—the values of all the properties of the individual can be prefetched by accessing one row of a table. The execution environment can cache the results and return them again to the application without issuing additional queries to the data source. Some cache invalidation mechanism must be employed in order to prevent the delivery of stale data when the data in the underlying data source is updated.

5.6 Value Propagation for Query Prefetching

In this section we describe the optimization we called Value Propagation for Query Prefetching. The optimization is based on the ability of the compiler to perform value propagation analysis and calculate known values of variables used in SPARQL queries. The compiler can produce code that will
prefetch the query results based on the calculated values of the variables.

For example, consider the code in Listing 5.14.

Listing 5.14: Using a variable with known values inside a SPARQL query

```python
id = 1
...
# some long portion of code that does not
# change the 'id' variable
...
if (someCondition) {
    id = 2
}
...
# SPARQL query that uses the 'id' variable
employee = select q::emp where {
    q::emp ont::empID id
}
```

In the code in Listing 5.14, the variable `id` is set on Line 1 to 1 and on Line 7 to 2, depending on satisfaction of some condition. The `id` variable is used in the SPARQL query on Line 12. The compiler performs data flow analysis and discovers that the possible values of the variable `id` in the query are 1 and 2. The compiler produces a parallel prefetching code for two query results based on the calculated possible values. When Line 11 is being executed, the results for both possible values of the `id` variable are already in the cache of the execution environment. Here the tradeoffs between prefetching redundant results and the gain in the code speed must be estimated by the compiler, to decide when to apply this optimization.

5.7 Redundant Query Elimination by Reasoning

In this section we describe the optimization we called *Redundant Query Elimination by Reasoning*. The optimization is based on the ability of a SWOOM compiler to perform reasoning during the compilation, in order to calculate query results in compile time. When the compiler is aware of ontologies of the data sources, it can apply reasoning to perform such calculation. (The compiler has to determine that the ontologies will not be
changed during the code execution. For example, the data sources containing the ontologies can be declared as read-only.)

As an example of such static query calculation, consider the query in Listing 5.15:

Listing 5.15: Redundant SPARQL query

```sparql
person = select q::person where {
  q::person rdf::type ont::Male.
  q::person rdf::type ont::Female. }
```

The query returns all the persons in the data sources that are both Male and Female. If the compiler is aware of the ontology where the classes `ont:Male` and `ont:Female` are defined, and if the ontology states that these two classes are disjoint, the compiler can calculate the result of the query statically. In the case shown in Listing 5.15, an empty list will be returned by the query since according to the ontology there could be no individual that belongs to both `ont:Male` and `ont:Female` classes. Knowing the results of the query statically, the compiler can optimize the code following the query based on the known results of the query.

### 5.8 Value Propagation by Reasoning

In this section we describe the optimization we called Value Propagation by Reasoning. As with the Redundant Query Elimination by Reasoning optimization, the Value Propagation by Reasoning optimization can be performed by a SWOOM compiler when it is aware of the ontologies of the data sources and when it can verify that these ontologies will not be changed during the code execution. The compiler can calculate some values statically and propagate these values in the code for further optimizations.

To see an example of such calculation, consider the code example in Listing 5.16.

Listing 5.16: Value Propagation by Reasoning

```python
person = ont::John.ont::hasSister
...
if (person.rdf::type.include?(ont::Male)) {
  # perform some computation if the person is Male
}
The code on Line 1 assigns the RDF individual that represents a sister of individual \texttt{ont::John}, to variable \texttt{person}. While the compiler cannot know the actual value of the assigned RDF individual, it can infer some properties of the individual. The compiler can achieve this by applying reasoning based on the ontology where the property \texttt{ont:hasSister} is defined. If the ontology states that the range of the property is \texttt{ont:Female}, the compiler can infer that the assigned variable belongs to the \texttt{ont::Female} class. The compiler can propagate the value of the \texttt{rdf:class} property (the class of the RDF individual) further and use it in other optimizations.

In our example, the compiler knows that according to the ontology, classes \texttt{ont:Male} and \texttt{ont:Female} are disjoint. The compiler combines this knowledge with the previously discovered fact that the class of the variable \texttt{person} is \texttt{ont:Female} on Line 3 of Listing 5.16. The compiler can infer that the class of the variable \texttt{person} is not \texttt{ont:Male} on Line 3, so the condition on Line 3 will be false and the code inside the conditional statement will not be executed. Such code can be eliminated by the compiler (dead code elimination).

### 5.9 Summary

In this chapter we presented several optimizations that can be performed by a SWOOM compiler. These optimizations are unique to SWOOM, since they rely on features of both the OO and the Semantic Web models. A compiler of an OO language cannot perform such optimizations on a program that processes data on the Semantic Web, since an OO compiler is not aware of the Semantic Web features.

Applying optimizations that exploit features of the Semantic Web is one of the benefits of SWOOM. Since it is a hybrid between the OO model and the Semantic Web model, a compiler (which is an OO tool) can be applied to the artifacts of the Semantic Web.

We described the optimizations for demonstrating the benefits of the hybrid model. We implemented one of the optimizations, namely \textit{Aggregation Pushing}, in Ruby-S and measured the optimization results. We measured significant performance improvement (by a factor of 480). In addition, to
demonstrate optimization potential of other optimizations, we simulated manually *Conditional Pushing, Join Pushing* and *Bulk Update*. We simulated and measured these optimizations, because they have high optimization potential. The code being optimized is an iteration over data items whose number can be very large. The performance improvement ranges from a factor of 2 to a factor of 27. While such drastic performance improvements are partly due to slow execution of Ruby, the results indicate that significant performance improvements can be achieved when these optimizations are performed in other SWOOM languages.
Chapter 6

On the Effectiveness of Programming in Ruby-S

In this chapter we evaluate the effectiveness of programming in Ruby-S. We chose basic programming tasks and compared their implementation in Ruby-S to the implementation in plain Ruby, using the SPARQL Client\textsuperscript{1} library and ActiveRDF\cite{active_rdf}.

6.1 Accessing a Property

The first basic task we examined is accessing a property of an RDF individual. The following listings show accessing the property \texttt{foaf:name} of the RDF individual \texttt{acme:id1}. The code examples assume that the namespaces \texttt{foaf} and \texttt{acme} are defined previously. The namespaces are defined in a similar way in all the compared packages, so we do not include them in the comparison. We assume that the \texttt{foaf::name} property is functional, i.e. RDF individuals can have only a single value for this property.

Listing 6.1 shows an access to a property using the SPARQL Client library. The size of the code is 61 characters. (We count characters without counting white spaces.)

Note that the code returns the values of the property as an array. When the property has a single value, the array has size 1 and the value is in cell 0. By [:\text{n}] we designate an access to the variable “:\text{n}”. This variable is a

\textsuperscript{1}http://sparql.rubyforge.org/client/
placeholder for the value of the property, used in the \textit{where} method of the SPARQL Client library.

Listing 6.1: Accessing a property using the SPARQL Client library

```ruby
1 sparql.select.where([ACME::id1, FOAF::name, :n]).execute[0][:n]
```

Listing 6.2 presents an access to a property in ActiveRDF. The size of the code is 41 characters. The code assumes that a string variable \textit{acme} which holds the appropriate URI prefix is already defined. Note that in ActiveRDF an object representing the RDF individual must be explicitly created before the properties of the individual are accessed.

Listing 6.2: Accessing a property in ActiveRDF

```ruby
1 ACME::Employee.new(acme + 'id1').foaf::name
```

Listing 6.3 illustrates an access to a property in Ruby-S. The size of the code is 20 characters. Note that since we assume that the property is functional, a single value is returned.

Listing 6.3: Accessing a property in Ruby-S

```ruby
1 acme::id1.foaf::name
```

The listings demonstrate that accessing properties in Ruby-S is twice and thrice shorter and much simpler than in Ruby, using libraries for RDF.

### 6.2 Updating a Property

The next basic task we examined is updating an RDF property. In SPARQL/Update, to update a property of an individual requires the programmer to delete the previous values of the property and to insert a new value for the property. The SPARQL Client library and ActiveRDF do not provide support for SPARQL/Update, so we compared the code of Ruby-S with plain Ruby code. The SPARQL/Update statement can be performed in plain Ruby by executing an HTTP POST command with the statement as the request string and with the \textit{SPARQL endpoint} as the server address. We used the Ruby \textit{Net::HTTP} class to perform the POST.

105
Listing 6.4 shows the SPARQL/Update statement required to update the property foaf:name of the individual acme:id1 to have a New Name value. The statement is written in the original SPARQL/Update syntax. The size of the statement is 181 characters.

Listing 6.4: A SPARQL/Update statement.

```sparql
PREFIX acme: <http://www.acme.com/employees/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

MODIFY DELETE { acme:id1 foaf:name ?name }
INSERT { acme:id1 foaf:name "New Name" }
WHERE { acme:id1 foaf:name ?name }
```

In Listing 6.5 we show performing the update in plain Ruby, i.e an HTTP POST of the SPARQL statement shown in Listing 6.4. (Here we assumed the variable newName is used for the new value of the property.) The targetURI, acme and foaf variables are defined previously (otherwise the code would be longer, making the comparison more in favor of Ruby-S). The size of the code is 205 characters.

Listing 6.5: Performing the SPARQL/Update statement of Listing 6.4 in plain Ruby

```ruby
:: Net::HTTP.post_form(URI.parse(targetURI),
  { 'request',
    'PREFIX acme: ' + acme + ' PREFIX foaf: ' + foaf +
    ' MODIFY DELETE { acme:id1 foaf:name ?name }' +
    ' INSERT { acme:id1 foaf:name "' + newName +
    ' WHERE { acme:id1 foaf:name ?name }" }' })
```

Listing 6.6 presents the same update in Ruby-S. The variable newValue is used to represent the new value of the property. The size of the code is 28 characters.

Listing 6.6: Performing the SPARQL/Update statement of Listing 6.4 in Ruby-S

```ruby
acme::id1.foaf::name = newName
```
The listings demonstrate that updating values of properties in Ruby-S is much shorter (by a factor of 6) than in plain Ruby. It is also easy to see that the code in Ruby-S is more readable than the plain Ruby code.

### 6.3 Performing a SPARQL query

Another important basic task we examined is performing SPARQL queries. Listing 6.7 shows the original SPARQL query, which we implemented by using Ruby-S, and by using two other Ruby packages in the consequent listings. The namespace foaf is defined previously. The query returns all the RDF individuals whose name is “John”. The size of the query is 32 characters.

Listing 6.7: A SPARQL query that finds RDF individuals whose name is “John”

```java
SELECT ?s WHERE { ?s foaf:name "John" }
```

Listing 6.8 illustrates an implementation of the SPARQL query shown in Listing 6.7 using the SPARQL Client library. The size of the code is 51 characters.

Listing 6.8: Implementing the SPARQL query using the SPARQL Client library

```java
sparql.select.where([:s, FOAF::name, "John"]).execute
```

The implementation in ActiveRDF is shown in Listing 6.9. Here the size of the code is 58 characters.

Listing 6.9: Implementing the SPARQL query using ActiveRDF

```java
Query.new.distinct(:s).where(:s, FOAF::name, "John").execute
```

The Ruby-S implementation is presented in Listing 6.10. The size of the code is 37 characters.

Listing 6.10: Implementing the SPARQL query in Ruby-S

```java
select q::s where {q::s foaf::name "John"}
```
The code examples show that the query embedded in Ruby-S is shorter (37 characters) than the queries implemented using other packages (51 and 58 characters), and its size is almost identical to the original SPARQL query (32 characters).

When a SPARQL query refers to several properties of the same individual, the differences between Ruby-S and other approaches become larger in favor of Ruby-S. This is because for several properties of the same individual, “shortcuts” are available for SPARQL and for Ruby-S, while no such shortcuts exist in other packages we examined.

Listing 6.11 shows a SPARQL query which refers to several properties of the same individual. We implemented this query using the SPARQL Client library, ActiveRDF and Ruby-S in consequent listings. The query returns all the RDF individuals with title “Dr.”, whose name is “John”. Note that for two properties of the same individual a shortcut is used—the individual appears once, and its properties with their values appear following a semicolon. The size of the query is 48 characters.

Listing 6.11: A SPARQL query to find Dr. John

```
1 SELECT ?s WHERE {
2   ?s foaf:name "John";
3       foaf:title 'Dr.'
4 }  
```

Listing 6.12 presents an implementation of the SPARQL query shown in Listing 6.11 using the SPARQL Client library. The size of the code is 80 characters.

Listing 6.12: Implementing the SPARQL query using the SPARQL Client library

```
1 sparql.select.where([:s, FOAF::name, "John"]).
2   where([:s, FOAF::title, "Dr."]).execute 
```

Listing 6.13 shows an implementation of the SPARQL query provided in Listing 6.11 using ActiveRDF. The size of the code is 86 characters.

Listing 6.13: Implementing the SPARQL query in ActiveRDF

```
1 Query.new.distinct(:s).where(:s, FOAF::name, 'John').
2   where(:s, FOAF::title, 'Dr.').execute 
```
In Listing 6.14, we show an implementation of the SPARQL query presented in Listing 6.11 in Ruby-S. The size of the code is 54 characters. Note that in Ruby-S the same shortcut (with semicolon) is used as in the original SPARQL query.

Listing 6.14: Implementing the SPARQL query in Ruby-S

```ruby
1 select q::s where {q::s foaf::name "John";
2 foaf::title 'Dr.'}
```

Consider the above code examples. The queries that refer to a single property of the same RDF individual are implemented in Ruby-S using 27% less characters than when using the other Ruby packages. The queries that refer to two properties of the same individual and for which shortcuts are used, are implemented using 32% less characters than when using the other packages we examined. In addition, the code in Ruby-S seems to be easier to read and understand than in the other packages.

### 6.4 Reasoning and Property Attachment

Another task we implemented involves logical reasoning. The task is calculating siblings of a person, based on information about brothers and sisters. Consider the situation in which partial information is available about brother and sister relations between people. For example, when according to some information it is known that $X$ is a brother of $Y$, but it is not known that $Y$ is a brother or a sister of $X$. In an OO programming language such information can be represented by using objects of type `Person`. Such an object can contain lists of other objects representing known brothers and sisters of the person. A method to return the siblings of a person would calculate the transitive closure of the sibling relation, when two persons are siblings if one of them is a brother or a sister of another.

Listing 6.15 provides a definition and presents a usage of the class `Person` (Lines 1–63) The class contains lists of objects representing brothers and sisters of a person. The method `getSiblings` receives a list of objects representing people. The method returns the objects, out of the input list, that represent the siblings of the person (Lines 29–61). Lines 65–69 define 5 objects representing people. Lines 72–74 set brother/sister relations between the people. (The relations are defined partially. For example, $p2$ is
set as a sister of \( p1 \), but \( p1 \) is not set as a sister or a brother of \( p2 \).) The size of the method is 27 lines, 440 characters.

Listing 6.15: Calculating siblings in Ruby

```ruby
class Person
  attr_accessor :name;

  def initialize(name)
    @name = name
    @brothers = []
    @sisters = []
  end

  def to_s
    @name
  end

  def addBrother(brother)
    brothers << brother
  end

  def addSister(sister)
    sisters << sister
  end

  def brothers
    @brothers
  end

  def sisters
    @sisters
  end

  def getSiblings(candidates)
    resultSet = [self]
    candidates
  end
end
```

110
added = true

while added do
  added = false
  newSiblings = []
  candidates -= resultSet

  resultSet.each do |sibling|
    newSiblings += sibling.brothers + sibling.sisters
  end

  candidates.each do |candidate|
    resultSet.each do |sibling|
      if (candidate.brothers + candidate.sisters).include? sibling
        newSiblings << candidate
      end
    end
  end

  newSiblings.each do |sibling|
    unless resultSet.include? sibling
      resultSet << sibling
      added = true
    end
  end
end

resultSet

p1 = Person.new("John")
p2 = Person.new("Mary")
p3 = Person.new("Alex")
p4 = Person.new("Bob")
p5 = Person.new("Jack")

p1.addSister p2
p2.addBrother p3
p4.addSister p2

print "Siblings of #{p1}:", # p1, p2, p3, p4 are printed
print p1.getSiblings([p1, p2, p3, p4, p5])

The task of calculating siblings can be performed in Ruby-S by first defining the ontology shown in Listing 6.16. Once the ontology is defined and put into one of the federated data sources, the code in Listing 6.17 sets the ont:hasBrother and ont:hasSister relations between objects and simply returns the values of the property ont:hasSibling of one of the objects. An OWL reasoner plugged into the Ruby-S environment will calculate all the siblings of an object based on the ontology.

Listing 6.16: Ontology that defines hasSibling, hasBrother and hasSister properties

```ruby
@prefix ont: <http://www.acme.org/ontology#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.

ont:hasSibling a owl:TransitiveProperty ,
     owl:SymmetricProperty .
ont:hasSister a owl:TransitiveProperty ;
     rdfs:subPropertyOf ont:hasSibling .
ont:hasBrother a owl:TransitiveProperty ;
     rdfs:subPropertyOf ont:hasSibling .
```

Listing 6.17: Setting brother and sister relations and calculating siblings in Ruby-S

```
p1.ont::hasSister+=[p2]
```
The advantage of using reasoning in Ruby-S is that the brother, sister and sibling relations are defined by an ontology, in a declarative, high-level and logical way. No special code for calculation of transitive closure (as implemented in plain Ruby in Listing 6.15) is required to be written, debugged and maintained. The size of the ontology is 8 lines, 288 characters. In addition to being a more natural way to represent knowledge (brother, sister and sibling relations), the ontology is shorter to write than the plain Ruby code (35% less characters and 70% less lines).

Note that objects p1, p2, p3 and p4 can belong to any class, each object to a different class. This is because in Ruby-S properties can be attached to any class (see Subsection 4.2.7). The advantage of Ruby-S in this aspect is that brother, sister and sibling relations can be defined between objects of different classes such as Employee, Student, ChessPlayer, etc., which do not have to inherit from a common class such as Person. No special class for handling brother and sister relations, such as the Person class shown in Listing 6.15, is required. The properties can be attached to any objects and reasoning can be applied to any objects.

6.5 Discussion

We based SWOOM on principles that are considered beneficial according to various Software Engineering literature we referenced in Section 3.2. While these principles seem reasonable to us, they are hard to quantify and measure for practical reasons. In general, in Software Engineering research, it is hard to prove that some programming language or programming paradigm is more beneficial than others. A study could be conducted to take several groups of programmers and let them implement the same programming tasks using the programming languages being studied. Various software engineering
metrics, such as number of code lines, cyclomatic complexity, etc., explained by Fenton and Pfleeger [31], can be applied to software products produced by different groups using the programming technologies being compared. A pitfall in such a study, however, is the risk of having significant differences between the groups of programmers. One of the earliest studies of individual differences between programmers, performed by Sackman et al. [54], showed differences of factor 25 in coding and of factor 28 in debugging between the best and the worst performers.

Another pitfall is that Software Engineering is concerned with development of large programs, and usually the benefits of programming paradigms become apparent only on a large scale. Due to this reason a comparative study would involve significant costs, i.e. it would be very expensive to perform a study on teams of programmers involving large tasks. The studies of programming paradigms are better performed in real conditions, by real programming teams performing programming tasks according to different paradigms. However, it is still difficult to compare the paradigms this way, since the level of expertise of programmers, the working conditions, the motivation and other factors are hard to neutralize. In our case, since the Semantic Web is not yet a mainstream technology, it would be harder to find programmers who are familiar with both the Semantic Web technologies and Ruby.

Due to the above reasons, we decided not to perform the study by implementing large programming tasks, and instead to illustrate and compare the implementation of basic tasks, such as accessing RDF properties, or updating properties, using Ruby-S and other Ruby packages for RDF processing. We showed that all the basic tasks are simpler in Ruby-S than in the other packages. In addition, it can be seen that the code in Ruby-S is more succinct than in the other packages.
Chapter 7

Conclusion

In this work, we presented a solution to the OO-RDF impedance mismatch. We introduced a model which is a hybrid between the OO model and the Semantic Web data model. We called this model SWOOM. We defined the principles of SWOOM and explained why they are beneficial for programming applications over the Semantic Web. We illustrated these principles by developing an implementation of SWOOM over Ruby—Ruby-S. We implemented a language environment for Ruby-S as a federated information system. The system federates multiple heterogeneous data sources, mapped to RDF via adapters. We extended a popular Relational-to-RDF adapter, D2RQ, to handle SPARQL/Update statements for updating data in relational data sources.

We implemented Ruby-S by extending the Ruby programming language using metaprogramming capabilities of Ruby. This work demonstrates the simplicity of developing new programming languages in Ruby without changing the parser and the interpreter.

One of the benefits of the hybrid model is that programming language tools, such as compilers, can process simultaneously the objects of the language and the data items on the Semantic Web, and can exploit Semantic Web features such as queries and reasoning. To illustrate this benefit, we proposed and evaluated several compiler optimization techniques, unique to SWOOM. The performance improvements we measured ranged from a factor of 2 to a factor of 480. Our experiments show that optimizations tailored for SWOOM provide significant impact and it is insufficient to merely rely on the optimizations of an OO programming language, when processing
data on the Semantic Web.

To demonstrate the benefits of programming in Ruby-S, we evaluated code of several basic programming tasks. In all the cases we examined, the code implemented in Ruby-S was shorter and, in our opinion, more readable than equivalent code in Ruby.

Our experiments show that SWOOM can be beneficial for developing applications over the Semantic Web, in terms of performance and programmer productivity.

Future work includes integrating Semantic Web technologies with other programming languages and with other programming tools, for example, with Integrated Development Environments (IDEs). We hope that in the future, the Semantic Web will become the mainstream technology for processing information. Once that happens, we hope that the ideas expressed in this work will have a significant impact on the development of languages for programming applications over the Semantic Web.
Bibliography


[4] U.S. Congress. Senate. Hearing before Armed Services Committee. Defense authorization request for fiscal year 2011; the future years defense program; the 2011 quadrennial defense review (QDR); the 2011 ballistic missile defense review (BMDR); the “Don’t ask, don’t tell” policy, February 2010.


[7] Dean Allemang, Thomas Baker, Reza B’Far, John G. Breslin, Edward Curry, Stefan Decker, Andre Freitas, Roberto García, Rosa Gil, Kristen Harris, Steve Harris, Constantine H kondros, Bernadette Hyland,


[52] Yoav Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB ’92, pages


تشكיפות של,mid, הגירה ושכפול של נתונים–המתכנת יכול לפתח תוכנה

 nelle-המתכנת, שאינו מתגורר בערivamente מעורר. תוכנה של

כותנים גילו סיבוב הריצת של Ruby-S. תכניות לעיבוד נתונים שרוחב

בלילהיות המודע למיקום הנתונים, בלילודעת אם הנתונים עוברים בין

כותנים מכוסכים על הנתונים הרגילים ב SKF עם קושי.

cלולות סיבובים של S-SPARQL תוכניות לעיבוד

 الأمري ידועים tócמות בהן התכונות של

ઇידיהים לברור, תוכן שיאפשר בכותנה של

ל公网安备ו תוכנית המוגשת עזר ו услуг תוכנה ב Ruby-S;

cויתות לאופטימיזציות ייחודיים, בכותן Shema, שמתור לא ש

וכל לתוך מודע לכותנה עם ב- SWOOM תוכנית בRuby-S, תוכנית על

אתה באופטימיזציות של. תוכנית המודר ב Ruby-S, תוכנית של

משובצות בכת מהפכה, ושכל תוכנית את המודע לכותנה ב- SWOOM תוכניות

בכדי שונים המודע וה cio של שפת השפה ישיו אופטימיזציה של

 değiştir קור קרónico, וה aracı תוכנית לקוד

מהירויותฺRuby-

אנו מקימים שטחי פיתוח שכותנה

ם-היידיויקיניים shoes, שכתנות STREET-S-NS ש하겠습니다

取り, בושא הקולות לשגון מسودה, בשיווי ש

בעד זה יש שיווי וברון, משמעת אילן

שהשתתפות STREET-S-NS.
The thesis explores the integration of URIs and namespaces in the Ruby language and RDF for model creation. It introduces new methods in Ruby for object creation in the RDF language, allowing for the addition of properties and SPARQL. It also explains that users can query methods from within RDF statements, similar to embedded procedures RDF, allowing for the creation of models with URI support. Additionally, the thesis mentions the use of SQL (through stored procedures) and allows for writing Ruby-SQL mixtures within SPARQL, allowing for SPARQL integration.

The thesis also describes the development of a framework for integrating information from various sources. These sources can be XML files, databases, and RDF, using D2RQ to create a single RDF model for data integration. The framework allows for probabilistic RDF representation and the update of RDF models using SPARQL.

The thesis concludes with the following key points:

- The thesis code and models are available for download.
- The thesis is a M.Sc. thesis from the Computer Science Department at Technion.
- The thesis was supervised by Prof. Amnon Shashar and Prof. Sagi Cohen.
- The thesis was completed in 2012.
כדי לפתח את הביעת חוסר ההתאמה, הענין מודל היברידי BIN המודל של רשת SWOOM הוא תכונת הסמיטית בינו מודל מונחה העצמים. קזרוד לס المتوו הצוותين במודל רשת SWOOM - Semantic Web Object Oriented Model.

הסוציאלי של מודל העצמים בינו מודל מונחה העצמים. קזרוד לס المتوו הצוותים מצד שנמצא מודל רשת SWOOM Semantic Web Object Oriented Model. העצמים של תכונת העצמים, העצמים של רשת SWOOM באטiconductor, תכונת ושימוש禘 של הרשת האיסנeteria ב.predicate של המודל של רשת SWOOM.

יה ordinances שפה התכתנית, ותעויות ושימוש禘 של ה聞いたו מתבנאת אתCelebration הקולנועים. המודל אחד ממגוון החברה של מודל רשת SWOOM באטiconductor, תכונת ושימוש婪 של הרשת האיסנeteria ב.predicate של המודל של רשת SWOOM.

יה ordinances שפה התכתנית, ותעויות ושימוש婪 של ה聞いたו מתבנאת אתCelebration הקולנועים. המודל אחד ממגוון החברה של מודל רשת SWOOM באטiconductor, תכונת ושימוש婪 של הרשת האיסנeteria ב.predicate של המודל של רשת SWOOM.

יה ordinances שפה התכתנית, ותעויות ושימוש婪 של ה聞いたו מתבנאת אתCelebration הקולנועים. המודל אחד ממגוון公社 של מודל רשת SWOOM Semantic Web Object Oriented Model.

בי לי שפת התכתנית, ותעויות ושימוש婪 של ה聞いたו מתבנאת אתCelebration הקולנועים. המודל אחד ממגוון公社 של מודל רשת SWOOM Semantic Web Object Oriented Model.

בי לי שפת התכתנית, ותעויות ושימוש婪 של ה聞いたו מתבנאת אתCelebration הקולנועים. המודל אחד ממגוון公社 של מודל רשת SWOOM Semantic Web Object Oriented Model.
תקציר
עיבודיעילשלמידעהואאחתהמשימותהחשובותבהנדסת תוכנה.ל棹, 
אחתהבעיות הקשות בפיתוח תוכנה לעיבוד מידה היא חוסר 
התאמה בין מודל הנתונים למודל השפה, כגון איה או תאמות בין 
מודל ר巼ס, RDFS, RDF–(.בעודשה(tecnologies של רשת ה 
אינטרנט הסמנטית OO) עצמים–יכולות להקלעל עיבוד מידע, 
ובפרטיה האינטגרציה של נתונים, SPARQL, OWL 
בעיה חוסר התאמה, שנובעת משונימהותibernהמודל של רשת ה 
אינטרנט הסמנטית (OO) לינוモデルמונחה עצמים, 
נותרה בעיה חוסר התאמה. 
הבדלים在这タイム זרים גורמים לשונימהותibernהמודל של רשת ה 
אינטרנט הסמנטית (OO) לינוモデルמונחה עצמים, 
כל עצם ישזהות מקומית, שנקבעת בזמן ריצה.ב–
המהווה המזהה URI (מזוהה ע”י SW אינדיבידואל) המקבילה 
של עצם ב–קבועי ייחודי bakım גלובלי. 
, אינדיבידואלים הם מתמידים)משיכים SW, העצמים הם 
זמןיים.ב–
• להתקיים אחרירית התכנית (והם נשמרים במקור נתונים 
של שום).אין מקבילה ל라도 SW, לעומזאת יש תכונות. 
ב–
, עצמים לאיכולים לשנות את המחלקות שלם.לעומזאת, 
OO בת أيام, אינדיבידואלים ב–
• אינדיבידואל יכול להיות שייך למחלקה יותר (100). 
 עצמים לאיכולים לשנות את המחלקות שלם באフリー. 
ב–
, עצמים לאיכולים לשנות את המחלקות שלם 
אינדיבידואלים ב–
• אינדיבידואל יכול להיות שייך למחלקה יותר (100).
המחקר נערך בהנחייה של ד”ר ירון קנזה בפקולטה למדעי המחשב.

אני מודע lettuce על התמיכה והסיום הנדרשים להשלמתו.
הבינונת ישモノים מעל רשת האינטרנט

חוקר על מחקר

لسומány הילני של חברה לעזרת להגירה של החזות
מניטר למטרות במודיעי המחשב

זיו איזנברג

הנהל לכסת הטכנולוגיה – מכון טכנולוגי לישראל
אלון תש״ע ב 2012 ספטמבר
תבניות יישומיים מעל רשת האינטרנט

הסמנה

ודיס איזנברג