

Chapter 11

Embedded Languages

Abstract The state of the art in programming Semantic Web applications is using complex application programming interfaces of Semantic Web frameworks. Extensive tests are necessary for the detection of errors, although many types of errors could be detected already at compile time. In this chapter, we propose an embedding of Semantic Web languages into the java programming language, such that Semantic Web data and queries are easily integrated into the program code, type safety is guaranteed, and already at compile time, syntax errors of Semantic Web data and queries are reported and unsatisfiable queries are detected.

11.1 Motivation

Programming Semantic Web applications requires to first learn a complex application programming interface of a Semantic Web framework like the one from Jena (Wilkinson et al. 2003). Furthermore, syntax errors in data files and in queries are only detected when the corresponding program instructions are executed, and error messages are reported from the Semantic Web framework. Therefore, extensive tests are necessary for developing stable Semantic Web programs, which must consider as much as possible from every branch in a program execution and from every possible input data. With our contribution, we want to address the following types of errors by a tool for embedding Semantic Web languages into programming languages, which use a static program analysis for avoiding or detecting these types of errors and therefore support the development of more stable programs: Queries with semantic errors do not generate error messages, but lead to unexpected behaviour of the Semantic Web application. Note that queries with semantic errors often return the *empty* result set every time, that is, these queries are *unsatisfiable*, which is a hint for a semantic error in the query. Furthermore, query results often contain data with a certain data type, for example, numeric values, which have to be further processed in data-type-dependent operations such as the summation of numeric values. Therefore, a cast is necessary to a specific programming language-dependant type. If an erroneous query contains values of other types than expected, then cast errors might occur at runtime.

So far suitable tools for embedding Semantic Web data and query languages into existing programming languages, which go beyond the simple use of application programming interfaces and take advantages of an additional program analysis at compile time, are missing. All static program analysis for *detecting errors* in the embedded languages can be processed *at compile time* before the application is really executed by starting the precompiler offering the most possible convenience to the programmer: Some tests may be done without using embedded languages, but then have to be done by using additional tools and copying and pasting code fragments, or by executing the program itself. A static program analysis can detect errors, which – without a static program analysis – may only be detected after running a huge amount of test cases, as the static program analysis considers every branch in the application code. Our tool, which we call *Semantic Web Objects* system (SWOBE), embeds

- The Semantic Web data language RDF/XML
- The query language SPARQL
- The update language SPARUL

into the *java* programming language. SWOBE supports the development of *more stable* Semantic Web applications by

- Providing transparent usage of Semantic Web data and query languages without requiring users to have a deep knowledge of application programming interfaces of Semantic Web frameworks
- Checking the syntax of the embedded languages for the detection of syntax errors already at compile time
- A static type check of embedded data constructs for guaranteeing type safety
- A satisfiability test of embedded queries for the detection of semantic errors in the embedded queries already at compile time
- A determination of the types of query results for guaranteeing type safety and thus avoiding cast errors.

A demonstration of the SWOBE precompiler is available online at [Groppe and Neumann \(2008\)](#), the example SWOBE programs of which cover embedding of RDF/XML constructs [see Assistant.swb, Student.swb, University.swb and Professor.swb at [Groppe and Neumann \(2008\)](#)], SPARQL queries (see TestStudent.swb and QueryTest.swb), and SPARUL queries (see Assistant.swb, Student.swb, University.swb, Benchmark.swb, Professor.swb, and UpdateTest.swb).

11.2 Related Work

We divide contributions to embedded languages into two groups: the embedding of relational query languages such as SQL into programming languages, and the embedding of XML data and its query languages into programming languages.

(continued)

Pascal/R is the first approach for a type safe embedding of a relational query language into a programming language (Schmidt 1977). Pascal/R does not use SQL but a proprietary query language.

There are many contributions that deal with the embedding of SQL into the programming languages C (see IBM 2003; Ingres 2006), Java (see ANSI 1998; Sybase 1998a; Erdmann 2002), Cobol (see Gilmore et al. 1994; Ingres 2006), Ada (see Erdmann 2002; Ingres 2006), Fortran (see Ingres 2006), Basic (see Ingres 2006), Pascal (see Ingres 2006), and functional programming languages (see Buneman and Ogori 1996; Wallace and Runciman 1999; Nagy and Stansifer 2006). Especially, Buneman and Ogori (1996), Bussche and Waller (1999), and Nagy and Stansifer (2006) describe a comprehensive type systems for guaranteeing static type safety for the relational algebra.

Kempa and Linnemann (2003) embed XML and the XPath query language into the object-oriented language Java and call the resultant language XOBJE (XML Objects). The generated XML data of an XOBJE-program are statically checked for type safety with respect to a given schema and the result types of embedded XPath queries are inferred. XOBJE_{DBPL} (Schuhart and Linnemann 2005) extends XOBJE with a transparent, type-independent, and distributed persistence-mechanism and a statically checked embedded update query language.

XDuce (Hosoya et al. 2005), HaXML (Wallace and Runciman 1999), and XMLambda (Shields and Meijer 2001) deal with the embedding of XML data into functional programming languages with special regard to parametric polymorphism in type systems for XML. <bigwig> (Brabrand et al. 2002) and JWG (Christensen et al. 2003) allow the embedding of static validated XHTML 1.0, the XML variant of HTML, into C and Java. Furthermore, Christensen et al. (2003) introduce high-level constructs for Web Service programming with an explicit session model.

Serfiotis et al. (2005) describe algorithms for a containment tester and the minimization of RDF/S query patterns. Serfiotis et al. (2005) consider a hierarchy of concepts and properties.

This chapter contains the contributions for embedding Semantic Web data and query languages into existing programming languages of Groppe et al. (2009e).

11.3 Embedding Semantic Web Languages Into JAVA

We first provide an overview of SWOBE and demonstrate the features of SWOBE by an example. Afterward, we explain the ideas and concepts of SWOBE in detail in the following subsections. We refer the interested reader to the specifications of RDF/XML (Beckett 2004), SPARQL (Prud'hommeaux and Seaborne 2008), and SPARUL (Seaborne and Manjunath 2008) for an introduction to the embedded data, query, and update languages.

```

(1) class TestStudent {
(2)   prefix rdf = http://www.w3.org/1999/02/22-rdf-syntax-ns#;
(3)   prefix ub = http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#;
(4)   prefix xsd = http://www.w3.org/2001/XMLSchema#;
(5)   prefix uni = http://www.University.edu/;
(6)   type Course = (URI, rdf:type,ub:Course) ∪ (URI,ub:name,string) ∪
(7)                 (URI,ub:shortName,string)? ∪ (URI, ub:date,date)+;
(8)   type Student = (URI,rdf:type,ub:GraduateStudent) ∪ (URI, ub:name,string) ∪
(9)                 (URI,ub:takesCourse,URI) ∪ Course+ ∪
(10)                ((URI,ub:emailAddress, anyURI) | (URI, ub:telephone, integer))*;
(11) type Students = Student+;
(12) public int[] getTelephoneNumbers(){
(13)   String courseURI = "uni:Programming";
(14)   int tel = 0565664732; int tel2 = 0565457893;
(15)   String email = "Henry@hotmail.com";
(16)   rdf<Course> course = <ub:Course rdf:about=#courseURI#>
(17)                 <ub:name rdf:datatype="xsd:string">Programming</ub:name>
(18)                 <ub:shortName rdf:datatype="xsd:string">Prog</ub:shortName>
(19)                 <ub:date rdf:datatype="xsd:date">2008-02-09</ub:date>
(20)                 <ub:date rdf:datatype="xsd:date">2008-02-16</ub:date>
(21)                 </ub:Course>;
(22) rdf<Students> students = <rdf:RDF><ub:GraduateStudent rdf:about="uni:MatrNr552662">
(23)                 <ub:name rdf:datatype="xsd:string">Henry Schmidt</ub:name>
(24)                 <ub:takesCourse>#course#</ub:takesCourse>
(25)                 <ub:telephone rdf:datatype="xsd:integer">#tel#</ub:telephone>
(26)                 <ub:emailAddress rdf:datatype="xsd:anyURI">#email#</ub:emailAddress>
(27)                 </ub:GraduateStudent>
(28)                 <ub:GraduateStudent rdf:about="uni:MatrNr552663">
(29)                 <ub:name rdf:datatype="xsd:string">Anne Mustermann</ub:name>
(30)                 <ub:takesCourse>#course#</ub:takesCourse>
(31)                 <ub:telephone rdf:datatype="xsd:integer">#tel2#</ub:telephone>
(32)                 </ub:GraduateStudent> </rdf:RDF>;
(33) SparqlIterator query = sparql ( SELECT ?Y ?Z FROM #students#
(34)                               WHERE {
(35)                                   ?X rdf:type ub:GraduateStudent .
(36)                                   ?X ub:telephone ?Y .
(37)                                   ?X ub:takesCourse ?V .
(38)                                   { ?V ub:shortName ?Z . }
(39)                               UNION
(40)                                   { ?V ub:name ?Z . } } );
(41) int[] telnumber = new int[query.getRowNumber()];
(42) String[] name = new String[query.getRowNumber()];
(43) int i = 0;
(44) while(query.hasNext()){
(45)   Result res = query.next();
(46)   telnumber[i] = res.getY();
(47)   if(res.getZ() != null) name[i] = res.getZ();
(48)   i++;
(49) query.close();
(50) return telnumber; } }

```

Fig. 11.1 A SWOBE example program. The bold facepart contains specific SWOBE expressions

Figure 11.1 contains an example SWOBE program, which uses the RDF format to describe information about students and the courses they take [see lines (16–32) of Fig. 11.1]. The type of the embedded RDF data is defined in lines (6–11) of Fig. 11.1. An embedded SPARQL query [see lines (33–39) of Fig. 11.1] asks for the telephone number of those students, which take at least one course. Additionally, the name and the short name of the courses taken by the student are contained in the query result. Afterward, the telephone numbers of the students and the names or short names respectively of the courses are stored in arrays by iterating through the query result [see lines (41–48) of Fig. 11.1].

Figure 11.2 depicts the architecture of the SWOBE precompiler.

The SWOBE precompiler first parses the SWOBE program according to the Java 1.6 grammar with the extension of embedded RDF/XML constructs (e.g., lines (16–21) and lines (22–32) of Fig. 11.1), prefix declarations (e.g., lines (2–5) of Fig. 11.1), and SPARQL/-UL queries (e.g., the SPARQL query in lines (33–39) of Fig. 11.1). Syntax errors of embedded RDF/XML constructs and SPARQL/-UL queries are already detected and reported in this phase at compile time, which are otherwise – in the case of using Java 1.6 with Semantic Web application programming interfaces – only detected at runtime maybe after running extensive tests.

We assume *S* to be the type of the right side of an assignment of RDF data to a variable (lines (16–32) of Fig. 11.1) and *T* to be the variable type. The type system of the SWOBE precompiler then checks whether or not *S* conforms to *T*, that is, if *S* is a subtype of *T*.

The satisfiability tester of the SWOBE precompiler afterward checks whether the result of the embedded SPARQL/-UL queries (e.g., lines (33–39) of Fig. 11.1 for a SPARQL query) is empty for any input based on the type of the input data.

The SWOBE precompiler then determines the java types of the results of the embedded SPARQL queries. The SWOBE precompiler uses the java types for

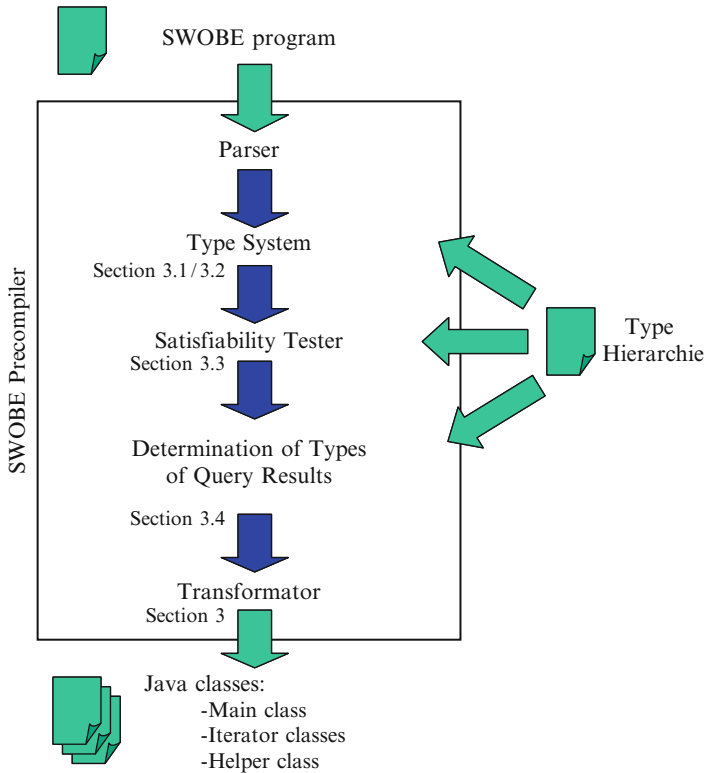


Fig. 11.2 The architecture of the SWOBE precompiler

generating the special iterators for query results. In the example of Figure 11.1, the SWOBE precompiler generates an iterator for the result of the SPARQL query in lines (33–39). The iterator contains the special methods `int getY()` and `String getZ()` for accessing the results of the variables `Y` and `Z`, respectively. Note that the SWOBE precompiler also determines the result type of `Y` to be `int` and of `Z` to be `String` based on the type `Students` of the input data `#students#` of the SPARQL query.

At last the SWOBE precompiler transforms the SWOBE program into java classes. The generated java classes use the application programming interfaces (API) of existing Semantic Web frameworks. Our SWOBE precompiler currently supports the API of the widely used Jena Semantic Web framework (Wilkinson et al. 2003), but can be easily tailored to support APIs of other Semantic Web frameworks. The transformed java classes are the main class corresponding to the SWOBE program, some iterator classes for query results like query in Fig. 11.1, and helper classes, methods of which are called by the main class.

11.3.1 The Type System

RDFS/OWL ontologies are designed to handle incomplete information. This design decision leads to the following two phenomena:

1. OWL and RDFS ontologies do not provide any constraints for entities that are not typed. For example, the triple $(s, \text{rdf:type}, c)$ types the s entity to be of class c . If an ontology is given, which has constraints for members of the class c like a maximal cardinality one of a property `color` for entities of the class c , then the triples $(s, \text{color}, \text{blue})$ and $(s, \text{color}, \text{red})$ are inconsistent with this ontology. However, no entity, not `_:b1` and not `_:b2`, is typed in the triple set $\{(\text{_:b1}, \text{uni:name}, \text{"UniversityOfLübeck"}), (\text{_:b1}, \text{uni:institute}, \text{_:b2}), (\text{_:b2}, \text{uni:name}, \text{"IFIS"})\}$, such that no any ontology can impose constraints on the triples of this triple set. Thus, this triple set conforms to any ontology.
2. Even if an entity is typed, a given ontology does not impose any constraints for properties and objects, which are not listed in the ontology. Thus, a fact (s, p, o) is still consistent with a given RDFS/OWL ontology, even when $(s, \text{rdf:type}, c)$ is true and there are no constraints given in the RDFS/OWL ontology about the object o or predicate p for members of the class c .

However, if the types S and T are described by ontologies, the check whether or not a type S is a subtype of another type T would not consider the triples not described by an ontology according to phenomena (1) and (2). In this case, we could only state that S is a subtype of T *except* of triples according to phenomena (1) and (2). Additionally, the satisfiability test of embedded SPARQL queries based on a given type for the input triples would detect only *maybe* unsatisfiable queries, which are unsatisfiable for triples without those of phenomena (1) and (2). However, we cannot guarantee the exclusion of triples according to phenomena (1) and

```

TypeDefinition ::= OrType | "ANY"
OrType ::= UnionType ( "I" UnionType )*
UnionType ::= TripleExpr ("+"|"*"|"?" )?
              ("U" TripleExpr ("+"|"*"|"?" )?)*
TripleExpr ::= ("(" ElemExpr "," ElemExpr "," ElemExpr ")")
              | "<IDENTIFIER>" | ("(" OrType ")")
ElemExpr ::= Value ( "\" ( Value | "<STRING_LITERAL>"
              ("I" "<STRING_LITERAL>"* )" ) )?

```

where <IDENTIFIER> represents an identifier (the name of a named type), Value a basic type and <STRING_LITERAL> a string literal.

Fig. 11.3 EBNF rules for defining types of embedded RDF data

(2). Furthermore, the determination of the query result types based on a given type for the input triples fails to consider the triples according to phenomena (1) and (2).

Therefore, we propose to use a type system, which avoids the two aforementioned phenomena of RDFS/OWL ontologies. Note that our type system supports incomplete information by allowing any triples if explicitly stated.

Our developed language for defining the types of embedded RDF data conforms to the EBNF rules of Fig. 11.3.

We can define the types of triple sets with this language. If the type is ANY, then there are no restrictions to the triple set. Types for single triples consist of three basic types for the subject, predicate, and object of a triple. A basic type is a concrete URI, literal, or an XML Schema data type. We can exclude values for basic types; for example, string \ “Fritz” allows all strings except “Fritz”. Furthermore, if A and B are types for triple sets, then $A \mid B$, $A \cup B$, A^* , A^+ , $A^?$ and (A) are also types for triple sets: $A \mid B$ allows triples of type A or B. A triple set V conforms to a type $A \cup B$ if triple sets V_1 and V_2 exist, such that $V_1 \cap V_2 = \{ \}$ and $V = V_1 \cup V_2$ and V_1 conforms to type A and V_2 conforms to type B. Arbitrary repetitions can be expressed by using A^* for including zero repetitions, A^+ for at least one repetition and $A^?$ for zero or one repetition. Bracketed expressions (A) allow specifying explicit priorities between the operators of a type, for example, $(A \cup B)^*$. References to named types may be used in a type for reusing already defined types.

We further allow types for predicate-object-lists, triples of which have the same subject, and for object-lists, triples of which have the same subject and the same predicate. We do not present these extensions here due to the simplicity of presentation, as these extensions complicate the algorithms for subtype tests due to more cases to be considered, but do not show new insights for subtype tests.

In the example of Fig. 11.1, the type definition Course [see lines (6–7) of Fig. 11.1] describes the RDF data about a course at a university. The type definition Student [see lines (8–10) of Fig. 11.1] describes the RDF data about a student in a university, and the type definition Students [see line (11) of Fig. 11.1] describes a group of students.

11.3.2 Subtype Test

Whenever a variable is assigned with RDF data [e.g., lines (22–32) of Fig. 11.1], we can determine the type of this assigned RDF data. The type S of assigned RDF data [e.g., the assigned RDF data in lines (22–32) of Fig. 11.1] is the union of the types of the triples, which are generated, and the types of embedded variables [e.g., `#course` in line (24)] containing RDF data. The subtype test is used for checking whether or not the type S of assigned RDF data [e.g., the assigned data in lines (22–32) of Fig. 11.1] conforms to an expected type T [e.g., the type `rdf<Students>` of the variable `students` in line (11) of Fig. 11.1] for the content of the assigned variable. In general, the subtype test checks whether or not a type S is a subtype of another type T ; that is, the subtype test checks whether or not all possible input data, which are of type S , are also of type T .

We first simplify the type definition T and S according to the formulas presented in Fig. 11.4, such that superfluous brackets are eliminated and subexpressions of the form $A \theta_1 \theta_2$, where $\theta_1, \theta_2 \in \{+, *, ?\}$, are transformed into subexpressions $A \theta_3$ with one frequency operator θ_3 .

The algorithm `checkSubType(S,T)` (see Fig. 11.5) performs the task of checking if S is a subtype of T . In the special case that the type S describes an empty triple set [see line (2) of Fig. 11.5], it is tested whether or not T allows the empty triple set by using the function `isNullable` (see Fig. 11.6). If T allows any input, then any type is a subtype of T [see line (3) of Fig. 11.5]. If T does not allow any input, but S does,

Fig. 11.4 Simplifying type definitions, where A is a type definition

$$\begin{aligned} A \theta_0 \theta_1 &= A^* \text{ if } \exists i \in \{0, 1\} \wedge \theta_i \in \{+, *\} \wedge \theta_{(i+1) \% 2} \in \{?, *\} \\ (A \theta_0) \theta_1 &= A \theta_0 \theta_1, \text{ where } \theta_0, \theta_1 \in \{+, *, ?\} \\ ((A)) &= (A) \\ A \theta \theta &= A \theta, \text{ where } \theta \in \{+, *, ?\} \end{aligned}$$

```
(1) boolean checkSubType(S, T) {
(2)   if(S = ∅) return isNullable(T);
(3)   else if(T = ANY) return true;
(4)   else if(S = ANY) return false;
(5)   else return (∃m = {(s1, t1), ..., (sn, tn)}, where si ∈ SExpr(S) ∧
                                     ti ∈ SExpr(T): homomorphism(S, T, m)); }
```

Fig. 11.5 Main Algorithm for the test if S is a subtype of T

$\text{isNullable}(A \mid B) = \text{isNullable}(A) \vee \text{isNullable}(B)$	$\text{isNullable}(A *) = \text{true}$
$\text{isNullable}(A \cup B) = \text{isNullable}(A) \wedge \text{isNullable}(B)$	$\text{isNullable}(A ?) = \text{true}$
$\text{isNullable}(A +) = \text{isNullable}(A)$	$\text{isNullable}((s, p, o)) = \text{false}$
$\text{isNullable}((A)) = \text{isNullable}(A)$	$\text{isNullable}(\emptyset) = \text{true}$

Fig. 11.6 Algorithm `isNullable`, where A and B are type definitions, s , p , and o the types of the subject, predicate, and object of a triple

then S cannot be a subtype of T [see line (4) of Fig. 11.5]. Otherwise, we first transform the types S and T into tree representations $\text{tree}(S)$ and $\text{tree}(T)$ by using a function tree . See Fig. 11.7 for the recursive definition of the function tree and an example of its result in Fig. 11.8.

Checking if S is a subtype of T can be reformulated into the problem of finding a homomorphism [see line (5) of Fig. 11.5] from the tree representation $\text{tree}(S)$ of S to the tree representation $\text{tree}(T)$ of T . We first describe an optimized algorithm for quickly finding such a homomorphism and afterward describe the constraints of the homomorphism in detail for the subtype relation between S and T . We use the

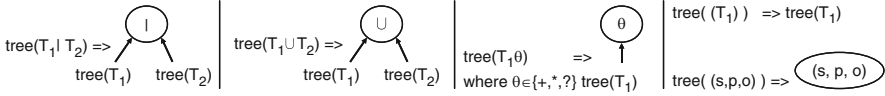


Fig. 11.7 Transforming a type definition into a tree representation. T_1 and T_2 represent type definitions and (s, p, o) the type of a triple

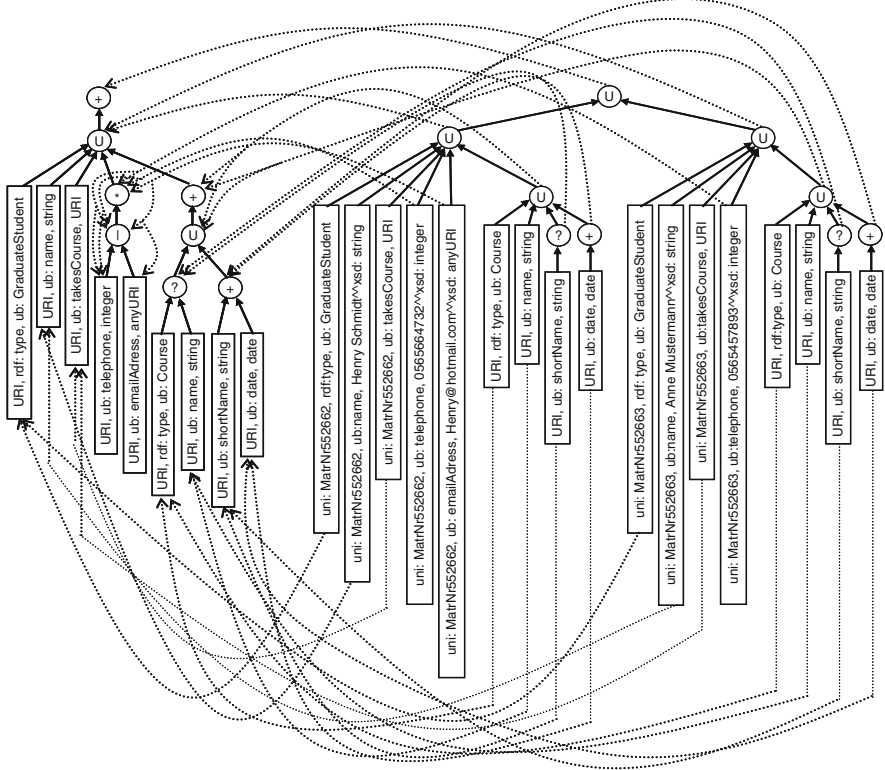


Fig. 11.8 Homomorphism from a type S representing the type of the assigned data in lines (22–32) of Fig. 11.1 on the right side of this figure to a type T representing the type $\text{rdf}\langle\text{Students}\rangle$ of the variable students in line (11) of Fig. 11.1 on the left side of this figure

tree representation of T and S and a corresponding homomorphism in the example of Fig. 11.8.

A subtype relation is already proved after only one homomorphism relation between two types is found.

We propose to search for a homomorphism between two types in two phases. The first phase determines all single candidate mappings from subexpressions of S to subexpressions of T. The candidate mappings can be determined very fast in the time $O(|S|*|T|)$, where $|V|$ represents the length of a type V, that is, the number of subexpressions of V. The efficient algorithm visits the tree representation of S bottom-up in order to find suitable subexpressions in T. During visiting S bottom-up, the algorithm considers already found mappings for the current node's children in the tree representation of S to nodes in the tree representation of T.

Afterward, we determine suitable subsets of the candidate mappings, which will be checked by the algorithm homomorphism (see Fig. 11.9) and this is explained in the next paragraph. This second phase is designed to quickly exclude unsuitable subsets of the candidate mappings. Furthermore, we first check those subsets of candidate mappings, which are promising to be a homomorphism. As a subexpression in S must not be mapped to two different subexpressions in T, we exclude this kind of subsets of the candidate mappings. In order to further exclude subsets of the candidate mappings earlier, we consider afterward candidate mappings for a subexpression in S, which is mapped to a minimum number of subexpressions in T in the remaining set of candidate mappings. In order to abort the search for a homomorphism in unsuitable subsets of the candidate mappings as early as possible, we do *not* consider a possible subset of candidate mappings further if a mapping (s, t) is in the subset C of candidate mappings, where one child s_1 of s and a mapping $(s_1, t_1) \in C$ exists such that $t \neq t_1$ and t_1 is not a subexpression of t. In this case, the constraints imposed by the homomorphism of the subtype relation cannot be fulfilled anymore.

The function homomorphism (see Fig. 11.9) expects the types T and S and a mapping m as input. The function first checks if a subexpression of S is mapped to

```

(1) boolean homomorphism(S, T, m) {
(2)   if(  $\exists s \in \text{SEExpr}(S): (s, t_1) \in m \wedge (s, t_2) \in m \wedge t_1 \neq t_2 \wedge$ 
(3)      $!(\exists t_1', \dots, t_n' \in \text{SEExpr}(T) : ((t_1' = t_1 \wedge t_n' = t_2) \vee (t_1' = t_2 \wedge t_n' = t_1)) \wedge$ 
(4)      $\forall i \in \{1, \dots, n-1\}: ((t_i' = t_{i+1}' \theta, \text{ where } \theta \in \{+, *, ?\}) \vee (t_i' = t_{i+1}' \mid t' \vee t_i' = t' \mid t_{i+1}',$ 
(5)      $\text{ where } t' \in \text{SEExpr}(T)) \vee (t_i' = t_{i+1}' \cup t' \vee t_i' = t' \cup t_{i+1}',$ 
(6)      $\text{ where isNullable}(t') \wedge t' \in \text{SEExpr}(T))))$  return false;
(7)   if(  $(S, T) \in m \wedge \forall s \in \text{SEExpr}(S): \exists (s, t) \in m \wedge$ 
(8)      $\forall (s, t) \in m: \text{isMappingOfHomomorphism}(s, t, T, m)$  ) return true;
(9)   else return false; }
```

Fig. 11.9 Function homomorphism for checking if m describes a homomorphism from S to T

only one subexpression of T [see line (2) of Fig. 11.9], as otherwise the mapping would be ambiguous with some exceptions: One exception is a subexpression with a frequency operator [see line (4) of Fig. 11.9]: if s is a subtype of t , then s is also a subtype of $t \theta$, where $\theta \in \{+, *, ?\}$. Another exception is a subexpression with an or-operator: if s is a subtype of t , then s is also a subtype of $t \mid t'$ [see line (4) of Fig. 11.9]. The last exception is a subexpression with a union-operator with at least one operand, which allows the empty expression [see line (5) of Fig. 11.9]: if s is a subtype of t , then s can be also a subtype of $t \cup t'$ with $\text{isNullable}(t)$ or $\text{isNullable}(t')$ holds and we later check whether or not s is really a subtype of $t \cup t'$. Afterward, the function homomorphism checks if the type S is mapped to the type T and if all subexpressions $\text{SEExpr}(S)$ (see Fig. 11.10) of S are mapped to subexpressions of T , fulfilling further constraints checked in the function $\text{isMappingOfHomomorphism}$ for each single mapping entry [see lines (7–8) of Fig. 11.9].

The function $\text{isMappingOfHomomorphism}$ checks if the given mapping m from type S to type T describes a part of a homomorphism from S to T . If S is composed of two subtypes S_1 and S_2 in an or-relation $S_1 \mid S_2$, then there should exist mappings from S_1 and S_2 to T [see line (2) of Fig. 11.11] for a subtype relation. If T is

$\text{SEExpr}(A_1 \mid \dots \mid A_n) = \{ (A_1 \mid \dots \mid A_n) \} \cup \text{SEExpr}(A_1) \cup \dots \cup \text{SEExpr}(A_n)$, where $n \geq 2$
 $\text{SEExpr}(A_1 \cup \dots \cup A_n) = \{ (A_1 \cup \dots \cup A_n) \} \cup \text{SEExpr}(A_1) \cup \dots \cup \text{SEExpr}(A_n)$, where $n \geq 2$
 $\text{SEExpr}(A \theta) = \{ (A \theta) \} \cup \text{SEExpr}(A)$, where $\theta \in \{+, *, ?\}$
 $\text{SEExpr}((s, p, o)) = \{ (s, p, o) \}$
 $\text{SEExpr}(A) = \text{SEExpr}(A)$

Fig. 11.10 Function SEExpr , where A, A_1, \dots, A_n are type definitions and (s, p, o) is the type of a triple

```

(1) boolean isMappingOfHomomorphism(s, t, T, m) {
(2)   if( $s = s_1 \mid s_2$ ) return  $((s_1, t) \in m) \wedge ((s_2, t) \in m)$ ;
(3)   else if( $t = t_1 \mid t_2$ ) return  $((s, t_1) \in m) \vee ((s, t_2) \in m)$ ;
(4)   else if( $s = s_1 \cup s_2 \cup \dots \cup s_n \wedge t = t_1 \cup t_2 \cup \dots \cup t_p$ ) {
(5)       return  $(\exists S'_1, \dots, S'_k: ((\text{repetition}(t, T) \wedge \exists q \in \mathbb{N}: k = p \cdot q) \vee (k = p))) \wedge$ 
(6)          $\forall i \in \{1, \dots, k\}: S'_i \subseteq \{s_1, \dots, s_n\} \wedge !(\exists j \in \{1, \dots, k\} - \{i\}: S'_i \cap S'_j \neq \emptyset) \wedge$ 
(7)          $((\cup_{S'_i \in S'_i} S'_i, t_{((i-1) \bmod p)+1}) \in m \vee (\text{isNullable}(t_{((i-1) \bmod p)+1}) \wedge S'_i = () \wedge$ 
(8)            $(\text{isNullable}(t_{((i-1) \bmod p)+1}) \vee S'_i \neq \emptyset)))$ ;
(9)   } else if( $s = (s_1, p_1, o_1) \wedge t = (s_2, p_2, o_2)$ ) {
(10)      return  $\text{isSubtype}(s_2, s_1) \wedge \text{isSubtype}(p_2, p_1) \wedge \text{isSubtype}(o_2, o_1)$ ;
(11)   } else if( $t = t_1 \theta_1$ , where  $\theta_1 \in \{+, *, ?\}$ ) {
(12)       if( $\text{freq}(s) \neq \text{freq}(t) \wedge !(\text{freq}(s) <_f \text{freq}(t))$ ) return false;
(13)       if( $s = s_1 \theta_2$ , where  $\theta_2 \in \{+, *, ?\}$ ) return  $((s_1, t_1) \in m)$ ;
(14)       else return  $((s, t_1) \in m)$ ; }

```

Fig. 11.11 Function $\text{isMappingOfHomomorphism}$ for checking if m describes a homomorphism from s to t

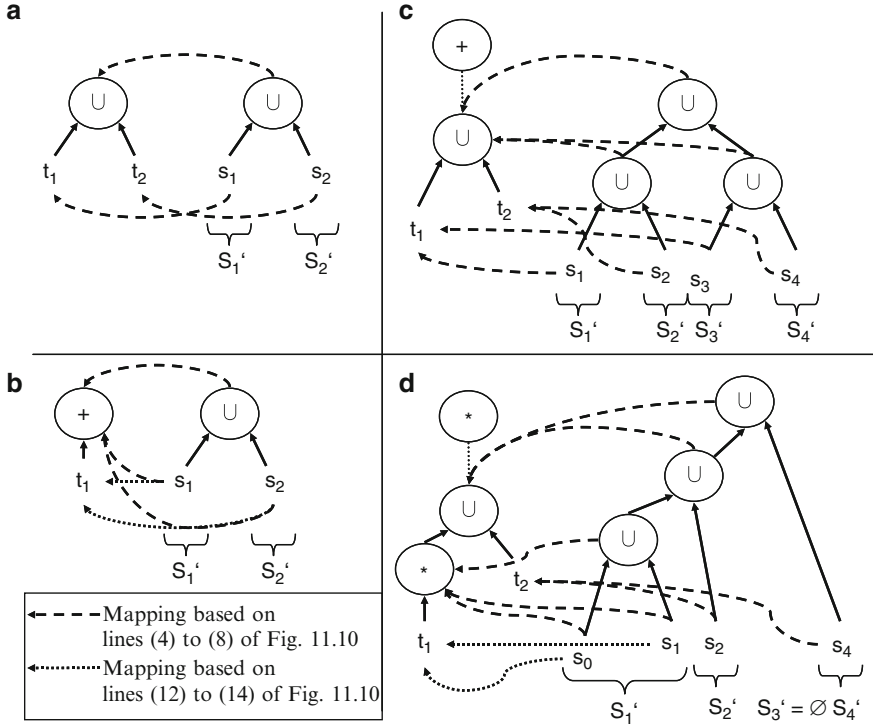


Fig. 11.12 Different examples for the subtype test when checking union operations. Here, t_1, \dots, t_p are subexpressions of T , s_1, \dots, s_n are subexpressions of S , S_1', \dots, S_k' are sets of the decomposition, s_i is a subtype of $t_{((i-1) \bmod p)+1}$, and s_0 is a subtype of t_1

composed of two subtypes T_1 and T_2 in an or-relation $T_1 \mid T_2$, then there should exist at least one mapping from S to T_1 or T_2 [see line (3) of Fig. 11.11] for a subtype relation.

We present examples for subtype tests between types containing union operations in Fig. 11.12. Example (a) in Fig. 11.12 is the simplest case. In this example, S and T are the union of two subtypes, and each union-operand of S must be a subtype of another union-operand of T . The pictures of the examples (b)–(d) in Fig. 11.12 become more complicated since the union-operator of T can be arbitrarily repeated (they are in the scope of a $*$ or a $+$ operator). In the example (b), several union-operands of S must be subtypes of the same union-operand of T . If T consists of a union of types, which can be arbitrarily repeated (see e.g., (c)), and the union-operator in S has more operands than the union-operator in T , then several pairwise disjoint decompositions of the union-operands of S must be subtypes of T . Furthermore, if an operand of the union-operator of T can be arbitrarily repeated like in example (d), then several union-operands of S can be the subtype of this arbitrarily repeatable union-operand of T . If an operand of the union-operator of T allows the empty triple set like in example (d), then no union-operand of S may be the

$$\text{repetition}(t, T) = (\exists t' \in \text{SEExpr}(T) : t \in \text{SEExpr}(t') \wedge t' = t'' \theta \wedge \theta \in \{+, *\})$$

Fig. 11.13 Function $\text{repetition}(t, T)$ for the determination whether or not the subexpression t is part of a subexpression of the type T , which can be arbitrarily repeated

Fig. 11.14 Function freq , where A and B are type definitions, s is the type of the subject, p of the predicate, and o of the object of a triple

$$\begin{aligned} \text{freq}(A \theta) &= \theta, \text{ where } \theta \in \{+, *, ?\} \\ \text{freq}(A \cup B) &= \text{One} \\ \text{freq}(A \mid B) &= \text{One} \\ \text{freq}((s, p, o)) &= \text{One} \end{aligned}$$

subtype of this union-operand of T . In order to deal with all these cases, we check the following condition: If S and T are composed of types in a union-relation [line (4) of Fig. 11.11], then there should exist a pairwise disjoint decomposition of the operands [line (6) of Fig. 11.11] of the union operator, such that the number of decompositions is the same as the number of union-operands of T or is a factor of the number of union-operands of T in the case that the subexpression of the union-operator can be arbitrarily repeated [line (5) of Fig. 11.11] as it is in the scope of a $*$ or a $+$ operator (see Fig. 11.13). Furthermore, all these decompositions must be mapped to the corresponding union-operand of T according to the number of repetitions [line (7) of Fig. 11.11] and all union-operands of T should have a candidate mapping or should allow the empty triple set [line (8); Fig. 11.11].

If the types S and T describe constraints for single triples, then each triple element, that is, the subject, predicate, and object, of S must be a subtype of the corresponding triple element of T [lines (9) and (10) of Fig. 11.11]; for example, `xsd:long` is a subtype of `xsd:decimal` according to the type hierarchy of XML Schema data types (see Peterson et al. (2009)). In the case that T contains an operator $+$, $*$ or $?$ [lines (11–14) of Fig. 11.11], we exclude those candidate mappings, the frequency of which is in conflict with a subtype relation in line (12) of Fig. 11.11. A mapping from S to T is not in conflict with a subtype relation, if they have the same frequency (see Fig. 11.14 for the computation of the frequency of a type) or if the frequency of S is lower than the frequency of T , that is, $\text{freq}(S) <_f \text{freq}(T)$, where the transitive relation $<_f$ holds for $\text{ONE} <_f ? <_f + <_f *$. Afterward, we check if the corresponding subexpressions are in the mapping m (lines (13 and 14) of Fig. 11.11).

11.3.3 Satisfiability Test of Embedded SPARQL and SPARUL Queries

Erroneous queries often return the empty set for any input, and thus queries are unsatisfiable. Therefore, an unsatisfiable query is a hint for errors in the query. Satisfiability tests of queries can (1) warn the user of the errors in queries, and debug SWOBE programs, thus leading to more stable programs, and (2) precompute

the unsatisfiable queries to the empty result at compile time, thus avoiding runtime processing and speeding up the program execution.

Note that SPARUL queries extend the syntax and semantics of SPARQL queries by update queries, such that the below described approaches apply to SPARQL queries *and* SPARUL queries. For checking the satisfiability of embedded queries, we first transform abbreviations of SPARQL/-UL constructs, that is, predicate-object-lists, object-lists, collections, and the a operator, into their equivalent long forms (see Groppe et al. (2009d)). We replace blank nodes by the variables, which are not used somewhere else in the SPARQL/-UL query according to Gutierrez et al. (2004). After this step, each triple pattern of the SPARQL/-UL query has the form $e_1 e_2 e_3$., where e_i is an IRI, a literal (including string and numeric constants) or a variable.

We determine the type D of the input data of the embedded query by a static program analysis. In the example of Fig. 11.15, the satisfiability test and the determination of the query result types are for the embedded SPARQL query in lines (33–39) of Fig. 11.1. The determined type for the variable ?Y is integer and the determined type for the variable ?Z is string. A triple pattern $e_1 e_2 e_3$ is satisfiable, if the type D of the input data contains types of triples, which intersect with $e_1 e_2 e_3$.. We can determine all possible types of variables in the triple pattern by checking all types of triples in D, which intersect with the triple pattern $e_1 e_2 e_3$. Thus, we can use $\text{types}(e_1 e_2 e_3)$ to determine the types of the variables in triple patterns (see Fig. 11.16). If the set of variable types is empty, then this triple pattern is unsatisfiable.

The satisfiability of queries can be determined by the function types of Fig. 11.16. Note that $\text{sat}(\text{Expr}, \text{types}(A))$ is a satisfiability tester for Boolean expressions Expr under data type constraints $\text{types}(A)$ of the variables in Expr. Such a satisfiability tester $\text{sat}(\text{Expr}, \text{types}(A))$ has a high computational complexity (see Cook (1971)). However, the results without using such a satisfiability tester $\text{sat}(\text{Expr}, \text{types}(A))$ for FILTER expressions are typically quite well, such that the application of such a satisfiability tester can be avoided to speed up computation.

If the result of the function types contains the empty set for the types of at least one variable, then the SPARQL/-UL query is unsatisfiable and we can warn the user.

```

SELECT ?Y ?Z FROM #student#
WHERE {
  ?X rdf:type ub: GraduateStudent. } {(?X, {URI})}
  ?X ub:telephone?Y . } {(?X, {URI}), (?Y, {integer})} } {(?X, {URI}), } {(?X, {URI}), } {(?X, {URI}), }
  ?X ub:takesCourse?V . } {(?X, {URI}), (?V,{URI})} } {(?Y, {integer})} } {(?V, {URI}), } {(?V, {URI}), }
  { ?V ub:shortName?Z . } } {(?V, {URI}), (?Z, {string})} } {(?Y, {integer})} } {(?Y, {integer})} } {(?Y, {integer})}
  UNION
  { ?V ub:name?Z . } } {(?V, {URI}), } {(?Z, {string})} } {(?Z, {string})} } {(?Z, {string})}
}

```

Fig. 11.15 Example of the satisfiability test and the determination of the query result types for the embedded SPARQL query in lines (33–39) of Fig. 11.1

$$\begin{aligned}
&\text{intersect}(t1, t2) = (t1 \text{ is a variable} \vee t1 \text{ is subtype of } t2 \vee t2 \text{ is a subtype of } t1) \\
&\text{types}(e_1 e_2 e_3, (e_1', e_2', e_3')) = \{ (e_i, ST) \mid i \in \{1, 2, 3\} \wedge e_i \text{ is a variable} \wedge \\
&\quad ST = \{e_i'\} \wedge \text{intersect}(e_1, e_1') \wedge \text{intersect}(e_2, e_2') \wedge \text{intersect}(e_3, e_3') \} \\
&\text{types}(e_1 e_2 e_3) = \{ (e_i, ST) \mid i \in \{1, 2, 3\} \wedge e_i \text{ is a variable} \wedge \\
&\quad ST = \bigcup_{(e_1', e_2', e_3') \in D} \text{types}(e_1 e_2 e_3, (e_1', e_2', e_3')) \} \\
&\text{types}(A B) = \{ (e, ST) \mid (e, ST') \in \text{types}(A) \wedge (e, ST'') \in \text{types}(B) \wedge ST = \{t \mid t \text{ is a super type of} \\
&\quad t' \in ST' \wedge t \text{ is a super type of } t'' \in ST''\} \}, \text{ where } A \text{ and } B \text{ are group graph patterns} \\
&\text{types}(\{A\}) = \text{types}(A), \text{ where } A \text{ is a group graph pattern} \\
&\text{types}(A \text{ OPTIONAL } B) = \text{types}(A) \cup \{ (e, ST) \mid (e, ST') \notin \text{types}(A) \wedge (e, ST') \in \text{types}(B) \} \\
&\text{types}(A \text{ UNION } B) = \{ (e, ST) \mid (e, ST') \in \text{types}(A) \wedge (e, ST'') \in \text{types}(B) \wedge ST = ST' \cup ST'' \vee \\
&\quad ((e, ST') \in \text{types}(A) \wedge (e, ST'') \notin \text{types}(B)) \vee ((e, ST') \notin \text{types}(A) \wedge (e, ST'') \in \text{types}(B)) \} \\
&\text{types}(A \text{ FILTER}(\text{Expr})) = \begin{cases} \text{types}(A) & \text{if sat}(\text{Expr}, \text{types}(A)) \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 11.16 The function types determining the types of variables in a SPARQL query, and the function intersect checking if elements of a triple type and a triple pattern intersect. A and B are SPARQL subexpressions, and e_i is the type of an element of a triple type or of a triple pattern

11.3.4 Determination of the Query Result Types

We have already determined the possible types of the result of an embedded SPARQL query when testing the satisfiability. For returning the result by an iterator, we have to determine a super type of these possible types. This super type is then the return type for the iterator method.

We present in Fig. 11.15 the determination of the query result types for the embedded SPARQL query in lines (33–39) of Fig. 11.1. The type for the variable ?Y in the query result is integer and the type for the variable ?Z is string.

Once we have determined the return type, we can generate code for a query result iterator with this return type, such that the type system of java guarantees type safety for the usages of the result. In the example of Fig. 11.15, the java type for the variable ?Y is int and the java type of the variable ?Z is string.

11.4 Summary and Conclusions

We have proposed an approach to supporting the development of more stable Semantic Web applications by embedding the Semantic Web languages RDF/XML, SPARQL, and SPARUL into the java programming language.

Our Semantic Web *Objects* system (SWOBE) uses a static program analysis in order to guarantee type safety, detect unsatisfiable SPARQL/-UL queries, and determine the types of query results at compile time. In this way, we avoid runtime errors and unexpected behavior of the Semantic Web application. Our implementation of the SWOBE system shows the advantages of our approach as a programming tool.



<http://www.springer.com/978-3-642-19356-9>

Data Management and Query Processing in Semantic
Web Databases

Groppe, S.

2011, IX, 270 p., Hardcover

ISBN: 978-3-642-19356-9