# Never Mind the Semantic Gap: Modular, Lazy and Safe Loading of RDF Data

Eduard Kamburjan, Vidar Norstein Klungre, and Martin Giese

Department of Informatics, University of Oslo, Norway
{eduard,vidarkl,martingi}@ifi.uio.no

**Abstract.** Any attempt at a tight integration between semantic technologies and object oriented programming will invariably stumble over the gap between the two underlying object models. We illustrate how this *semantic gap* manifests from the point of view of data retrieval with SPARQL. We present a novel mechanism to load data from RDF knowledge graphs into object-oriented languages that gives static guarantees about the data access and modularly integrates the mapping between the program and the RDF view with the class definition in the program. This allows us to preserve the separation of concerns between the class system of RDF (geared towards domain modeling and data), and that of the program (geared towards typing and code reuse). Loading of RDF can be performed lazily, when required by the program, based on *query-futures* – subqueries that are only evaluated if and when the data is accessed. We formulate a Liskov principle for the mapping queries to characterize when they respect the subclass relation. Moreover, we provide tool support to detect when the user-provided mapping would cause the loading mechanisms to result in data structures that manifest the semantic gap.

## 1 Introduction

*Motivation.* Despite the important role that Semantic Web technologies can play in modern software applications, their integration with programming languages remains a challenge. The main challenge is the so-called *impedance mismatch* [1,2] or *semantic gap* [3] between the object model of RDF, geared towards data-driven tasks, and the object model of programming languages, geared towards typeability and modularity.

This impedance mismatch for RDF manifests when mapping RDF into the class system of the program: to load data from an RDF store, one executes some SPARQL query, manually traverses the results and creates objects on the go to perform computation later on. This turns data loading into a fragile, work-intensive and highly error-prone task: (a) There is no type safety mechanism. (b) The mapping between the OO class and the RDF pattern is not modular — the retrieving queries quickly grow in size and are overwhelming for the programmer. (c) While most endpoints support lazy iterators, they do not support lazy data structures within one answer: If the application is only interested in parts of the loaded data, but only decides so after the query is formulated (e.g., based

on prior answers) it is not possible to postpone loading. (d) Finally, depending on the data model and retrieving query, one node in the RDF graph may be mapped to different objects, if it occurs in several answers.

In this work, we investigate *lazy*, *modular* and *type-safe* loading of data from RDF graphs into an object-oriented (OO) programming language. Instead of fighting the impedance mismatch, we embrace it and keep it under control: we clearly describe the structure for which such a mapping can be defined and give static analyses to warn the programmer about unintuitive effects.

*Approach.* We solve these challenges by providing tool support to the programmer: Our mapping tightly couples OO classes with graph patterns in RDF using a SPARQL query per OO-class. This query describes how to *construct* an object from an RDF graph, it does not establish a perpetual link between RDF nodes and OO objects. Moving the mapping from the point where the data *is used* to the program point where the data structures *are defined* simplifies modeling and allows *reuse* of queries. Reuse, in turn, is critical for maintenance. Without reuse, maintenance is aggravated if multiple queries perform similar data retrieval tasks, but are scattered over the program or dynamically manipulated.

Using the query containment-based typing mechanism presented in [4], we directly address type safety and include support for OO inheritance: we always load the most specific class possible and give a static analysis that checks uniqueness of this class. Furthermore, we give a Liskov principle [5] to statically check whether the retrieval query of a subclass correctly refines the retrieval query of the superclass. We give another analysis to inform developers whether multiple objects constructed during data access correspond to the same RDF node.

Integrating the mapping into the OO class structure enables us to perform *lazy evaluation*. Lazy evaluation only retrieves data if the computation indeed requires it. This is implemented as follows: When loading a class, its query is automatically executed. If a class has a field of another class type, this second query is lazily evaluated: The field is initialized with a *future* [6,7]. A future is a placeholder that contains the inner SPARQL query that is only evaluated when the future is explicitly accessed during a computation.

It is crucial to our approach that we do *not* relate *concepts* of RDF and OO to each other directly, i.e., do not enforce a one-to-one correspondence between OO objects and RDF individuals. Systems that try to close this *semantic gap* [3] fail to address the fundamental differences in assumptions and modeling techniques in RDF/OWL and OO, most prominently the open-world assumption and multiple inheritance. Instead, we embrace the differences in modeling and give the programmer a systematic and safe way to close the gap specifically for their application. Thus, our system can be used to both relate the OO and RDF representation of some objects, but also as a type-safe container to load the results of queries, where such a relation is not desirable.

We give an informal example in Sec. 2 and preliminaries in Sec. 3. Modular mappings are described in Sec. 4 and inheritance in Sec. 5. Lazy loading is introduced in Sec. 6 and evaluated in Sec. 7. Related work is given in Sec. 8.

```
1 List<Nodes> it =
2 query("SELECT * WHERE { ?o :id ?id; :stamp ?stamp; :back ?w1; :front ?w2.
3       ?w1 :wheelId ?wId1; :stamp ?last1. ?w2 :wheelId ?wId2; :stamp ?last2.
4       FILTER(?wId1 != ?wId2}.");
5 Int i = it.next().get("id"); //dynamic cast to Int
6 Bike bike = new Bike(i, ...);
```

**Fig. 1.** Dynamic data access with SPARQL.

## 2 Running Example

Before we formalize our approach, we give an informal example that shows the targeted application. We do not present advanced features, such as typing and inheritance here. The example is given in the `LMOL` language introduced in Section 6, where the exact syntax and runtime semantics are introduced. Consider the following domain model about bikes and an application that loads all bikes into its class structure. At a later point, the application will then use the data to perform some computation on the wheels. This is illustrated in Fig. 1

$$\exists \texttt{hasWheel}.\top \sqsubseteq \texttt{Bike} \qquad \texttt{back}, \texttt{front} \sqsubseteq \texttt{hasWheel} \qquad \top \sqsubseteq \forall \texttt{id}.\texttt{Int}$$
$$\exists \texttt{id}.\top \sqcup \exists \texttt{stamp}.\top \sqsubseteq \texttt{Bike} \sqcup \exists \texttt{hasWheel}^{-1}.\ \texttt{Bike} \qquad \top \sqsubseteq \forall \texttt{stamp}.\texttt{Int}$$

We observe the following: (a) the data access in line 5 requires dynamic typing, (b) The connection between classes and data is established by a a query that is not modular w.r.t. nested classes, and (c) we may retrieve too much data: if the computation on a bike stops after the first wheel, then the second wheel should be not loaded in the first place. Note that it may not be known at the time the data is loaded which computations will be performed on it, thus, it is not always possible to adjust the query. Even when this is the case, it leads to the situation that several queries are manually optimized versions of the same data retrieval.

Our idea is twofold: We (1) annotate the class declaration with a SPARQL query to retrieve instances of the class, and (2) use futures for nested class structures. The following shows an annotated `Wheel` class and how to use it.

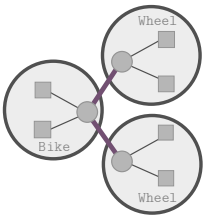
```
1 class Wheel anchor ?o
2   (Int wheelId, Int last) //id is the id of the wheel, not the IRI
3 end retrieve SELECT ?wheelId ?last
4              WHERE{ ?o :wheelId ?wheelId; :stamp ?last. }
5 ... List<Wheel> it = load Wheel();
```

There is *no* `Wheel` class declared in the RDF vocabulary – the annotated query models *retrieval*. The **load** statement returns an iterator over all results of the query (in some preconfigured KB). Note that already here, we introduce a separation of concerns in the language: data modeling is concentrated on the class, while computations can be performed with **load** as an encapsulation mechanism. Indeed, it is not visible to the programmer what kind of KB access is performed. The following code block shows the annotation for the `Bike` class.

```
1 class Bike anchor ?o(
2   Int id,  Int last, //id is the id of the bike, not the IRI
3   link(?o :front ?front) QFut<Wheel> front,
4   link(?o :back ?back) QFut<Wheel> back)
5 end retrieve SELECT ?id ?last WHERE { ?o :id ?id; :stamp ?last. }
6 ...
7   List<Bike> it = load Bike();  Bike bike = it[0];
8   List<Wheel> it = load bike.front;
9   Wheel w = it[0];
```

The `Wheel` instances are *not* described by the query. Their retrieval is already described in the `Wheel` class. The **link** annotation makes their loading lazy: When the **load** `Bike()` statement in line 8 is executed, only the query of `Bike` is executed. The fields of the `Wheel` class are initialized with *futures*: containers that contain the delayed query of `Wheel`. It is only in line 9, when `bike.front` is accessed, that the query below is executed. Note that it is enriched with information from the original query. This is crucial to correctly connect the nodes.

```
1 SELECT ?wheelId ?last WHERE{ run:obj1 :front ?w1.
2                              ?w1 :id ?wheelId; :last ?last. }
```




The annotations for lazy loading are illustrated in the figure to the right. The circles denote the parts of a graph retrieved by a single query and the thick edges are the **link** queries between the queries of the single classes. Lazy KB access aims to (1) make KBs more usable by reducing the load on the programmer and (2) allow a more flexible control over data loading by delaying the exploration of certain parts of the KB. The goal is not to replace all possible usages of queries and we stress that the lazy loading mechanism subtly changes the way the KB is accessed in two ways: First, by delaying the query, we take control away from the query planner and give it to the program. While this reduces the possibilities to optimize on the query level, it allows the programmer to be more flexible in its data modeling. Secondly, lazy evaluation may retrieve too many objects in the first step, as it is not known whether the next query will succeed or not. For example, the original query above will not return bikes with only one wheel, while lazy loading will do so and create a `Bike` object with one `Wheel` field evaluating to `null`. We address this by allowing to flatten the queries of nested classes to one overall query, but note that this is not possible for (mutually) recursive class structures.

## 3 Preliminaries

*Semantic Web.* We assume that the reader is familiar with the basics of established technologies of RDF KBs and SPARQL, and only repeat basic notation.

A knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is a pair of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$. We represent the ABox as a set of triples. A query is denoted as $\mathtt{Q}(\overline{x})$, where $x$ ranges over variables. Given a query $\mathtt{Q}(\overline{x})$, we say that $\overline{x}$ are its answer variables. The

| | | |
|---|---|---|
| $\mathsf{T} ::= \texttt{C} \mid \texttt{List<C>} \mid \texttt{Int} \mid \ldots$ | $\mathsf{Prog} ::= \overline{\mathsf{Class}}\ \textbf{main}\ \texttt{s}\ \textbf{end}$ | Types and Programs |
| $\mathsf{Class} ::= \textbf{class}\ \texttt{C}\ (\overline{\texttt{T f}})\ \textbf{end}$ | $\texttt{s} ::= \mathsf{T}\ \texttt{v:=rhs;} \mid \texttt{e.f:=rhs;} \mid \texttt{s s}$ | Classes and Statements |
| $\texttt{rhs} ::= \textbf{new}\ \texttt{C}(\overline{\texttt{f = e}}) \mid \texttt{e}$ | $\texttt{e} ::= \texttt{null} \mid \texttt{v} \mid \texttt{n} \mid \texttt{e.f} \mid \texttt{e[e]}$ | Expressions |

**Fig. 2.** Surface Syntax. The notation $\overline{\cdot}$ denotes lists, `f` ranges over fields, `v` over variables, `n` over literals and `C` over class names.

query $\mathtt{Q}$ may contain non-answer variables as well. If it has two answer variables, then we say that $\mathtt{Q}$ is binary. Given two queries $\mathtt{Q}_1, \mathtt{Q}_2$, we define their conjunction $\mathtt{Q}_1 \wedge \mathtt{Q}_2$ as the query that returns the intersection of answers to $\mathtt{Q}_1$ and $\mathtt{Q}_2$. We say that a query $\mathtt{Q}_1$ is contained in another query $\mathtt{Q}_2$ over a TBox $\mathcal{T}$ under an entailment regime $\mathsf{er}$, written $\mathtt{Q}_1 \sqsubseteq^{\mathcal{T}}_{\mathsf{er}} \mathtt{Q}_2$, if for every ABox $\mathcal{A}$ each answer to $\mathtt{Q}_1$ over $(\mathcal{T}, \mathcal{A})$ under $\mathsf{er}$ is also an answer to $\mathtt{Q}_2$ over $(\mathcal{T}, \mathcal{A})$ under $\mathsf{er}$. We say that two queries are equivalent, written $\mathtt{Q}_1 \equiv^{\mathcal{T}}_{\mathsf{er}} \mathtt{Q}_2$, if they contain each other. Furthermore, a query is said to be unsatisfiable under a TBox $\mathcal{T}$ if it has no possible answer under $\mathcal{T}$. Given two binary queries $\mathtt{Q}_1(?a, ?b)$, $\mathtt{Q}_2(?c, ?d)$ which have no variable in common, we define the concatenation $\mathtt{Q}_1(?a, ?b) \circ \mathtt{Q}_2(?c, ?d) = \mathtt{Q}_3(?a, ?d) = \mathtt{Q}_1(?a, ?x) \wedge \mathtt{Q}_2(?x, ?d)$. We say that a binary query $\mathtt{Q}_1(?a, ?b)$ is the inverse of another binary query $\mathtt{Q}_2(?a, ?b)$ if $\mathtt{Q}_1(?a, ?b) \equiv^{\mathcal{T}}_{\mathsf{er}} \mathtt{Q}_2(?b, ?a)$.

*Programming Model.* We use a minimal object-oriented programming language to illustrate our approach. While the implementation works on a full programming language with methods and additional statements, such as branching or loops, and the method itself can be adapted on top of any OO language, we give here only the minimal fragment to focus on the interaction between OO and RDF in a minimal setting. In this section, we give the basic structure of the language, while the later sections will extend it with a modular mapping (Sec. 4), inheritance (Sec. 5), and lazy loading (Sec. 6).

**Definition 1 (Syntax).** *The syntax of our base language is given in Fig. 2.*

A program is a set of classes, each defined by a set of fields and a name, and a main block, which is a sequence of statements. As statements we only consider assignments to fresh variables and fields, as well as object creation and sequence. For types we assume at least integers and parametric lists (to store the results of data loading). Expressions are standard, `e[e']` is list access.

The runtime semantics is defined using a standard Structured Operational Semantics (SOS) [8], i.e., a set of rewrite rules on runtime configurations. A runtime configuration contains the class table, the created objects and the statement that remains to execute. A rewrite rule takes a runtime configuration and transforms it by executing the next statement.

**Definition 2 (Runtime Semantics).** *A configuration* $\mathsf{conf}$ *is defined by:*

$$\mathsf{conf} ::= \big(\mathcal{K}, \mathsf{CT}\big)\ \mathtt{s}\ \big(\sigma, \mathsf{obs}\big) \qquad \mathsf{obs} ::= \mathbf{obj}(\mathtt{X}, \mathtt{C}, \rho) \mid \mathsf{obs}, \mathsf{obs}$$

*Where $\mathcal{K}$ is a knowledge base,* $\mathsf{CT}$ *is the class table, a map from class names to the set of their fields, $\sigma$ is a map from the variables in the main block to the literals or object identifiers,* $\mathtt{s}$ *is a statement or the special symbol $\epsilon$ for termination, and* $\mathsf{obs}$ *is a list of created objects. An object* $\mathbf{obj}(\mathtt{X}, \mathtt{C}, \rho)$ *has an identifier* $\mathtt{X}$*, a class* $\mathtt{c}$ *and a store $\rho$ that maps all fields of* $\mathtt{c}$ *to literals or object identifiers. We say that an object is* well-formed *if $\rho$ respects the annotated type, i.e., maps each field to a literal of the fitting data type or to an object of the correct class.*

Malformed objects can lead to undefined operations and our type system ensures *statically* that they do not occur at runtime.

An SOS-rule has the form $\mathsf{conf}_1 \to \mathsf{conf}_2$, optionally with some conditional premises that have to hold for the rule to be executable. Additionally, the function $[\![\cdot]\!]^{\sigma}_{\mathsf{obs}}$ evaluates an expression to a literal or object identifier, given a certain local store and a set of objects. We give only the rule for assignment of side-effect free expressions to local variables here, which we need in the following.

$$\textbf{(assign-local)}\ \frac{[\![\mathsf{e}]\!]^{\sigma}_{\mathsf{obs}} = l}{\left(\mathcal{K}, \mathsf{CT}\right)\ \mathtt{T\ v\ :=\ e;\ s}\ \left(\sigma, \mathsf{obs}\right) \to \left(\mathcal{K}, \mathsf{CT}\right)\ \mathtt{s}\ \left(\sigma[\mathtt{v} \mapsto l], \mathsf{obs}\right)}$$

## 4 Modular Loading

We extend the syntax with annotations that instantiate object instances from RDF graphs and add a statement to construct and execute the query for a class.

**Definition 3 (Syntax of MOL).** *The syntax of* MOL *is the one of Def. 1, with the rule for classes replaced, and the rule for expression extended by the following, where* $\mathtt{Q}$ *ranges over SPARQL queries and $x$ over SPARQL variables.*

$$\mathsf{Class} ::= \textbf{class}\ \mathtt{C}\ \textbf{anchor}\ x\ \left(\overline{[\textbf{link}\,(\mathtt{Q})]?\ \mathtt{T\ f}}\right)\ \textbf{end retrieve}\ \mathtt{Q} \qquad \mathsf{rhs} ::= \ \ldots \mid \textbf{load}\ \mathtt{c}()$$

The class definition now contains a query annotated with **retrieve**, which maps graph patterns to object instances. Additionally, it contains an **anchor** variable, which must occur in the **retrieve** query and is used to construct queries for nested classes. Finally, the **link** clause of each field of class-type links the graph pattern of this class with the graph pattern of the class of the field in question.

We write $\mathtt{anchor}_{\mathtt{C}}$ for the anchor variable of a class, $\mathtt{query}_{\mathtt{C}}$ for its **retrieve** query and $\mathtt{link}_{\mathtt{C},f}$ for the linking query. We assume the following, easy to check, syntactical restrictions: (1) $\mathtt{query}_{\mathtt{C}}$ has a connected graph pattern containing $\mathtt{anchor}_{\mathtt{C}}$ and one variable $?v_{\mathtt{f}}$ for each field $\mathtt{C.f}$ that has a data-type (2) $\mathtt{link}_{\mathtt{C},f}(\mathtt{anchor}_{\mathtt{C}}, ?v_{\mathtt{f}})$ is a binary SPARQL query with a connected graph pattern. The set of fields $\mathtt{f}$ with linking queries in $\mathtt{c}$ is denoted $\mathtt{cf}(\mathtt{c})$. For our mapping, all fields of the class, as well as the fields of objects referred to are mapped to one query variable. This is not possible in general: for example, consider **class** `Link` **anchor** `?o` (**link**`(?o :next ?x) Link x`) **end retrieve** `?o a :C` – there is no bound on the retrieved object, which can be an arbitrarily long list, so there is no a priori bound on the number on variables for the query. We can identify classes for which we can construct a finite query as *forward-cycle-free* in their structure.

A class is forward-cycle-free, if all cycles caused by link clauses between classes can be resolved by considering one of the clauses as the inverse of the others. Forward-cycle-freedom is only needed for eager queries.

**Definition 4 (Retrieval Trees and Forward-Cycle-Free Classes).** *Let $C$ be the set of classes in a program $P$ and $\mathfrak{G}(P) = (C, L)$ be its* retrieval tree *with edges $L \subseteq C \times Q \times C$. An edge $(C_1, \mathtt{Q}, C_2)$ is part of $L$, if $C_1$ has a field of type $C_2$ or* `List<`$C_2$`>` *with link query* `Q`*. Let $\mathfrak{G}(P, \mathtt{C})$ be the subgraph of $\mathfrak{G}(P)$ that is reachable from* `C`*.*

*Given a cycle $e_1, \ldots, e_n$, we say that edge $e_1$ is* backwards *if $\mathtt{Q}_1$ is the inverse of $\mathtt{Q}_2 \circ \cdots \circ \mathtt{Q}_n$.*[1] *We say that* `C` *is* forward-cycle-free *if every cycle in $\mathfrak{G}(P, \mathtt{C})$ contains a backwards edge that does not originate in* `C`*, and removal of all the backwards edges turns $\mathfrak{G}(P, \mathtt{C})$ into a directed acyclic graph, and does not make any node unreachable from* `C`*. We denote this graph with $\mathfrak{R}(P, \mathtt{C})$.*

We omit the $P$ parameter if the program is understood. We can now define the eager queries for forward-cycle-free classes.

**Definition 5 (Eager Queries for Forward-Cycle-Free Classes).** *The* eager query `eq(C)` *of a forward-cycle-free class* `C` *is defined as follows. We set*

$$\mathtt{eq}(\mathtt{C}) = \mathtt{query}_{\mathtt{C}} \wedge \bigwedge_{(\mathtt{f},\mathtt{D}) \in \mathtt{cf}(\mathtt{C})} \left( \mathtt{link}_{\mathtt{C},\mathtt{f}}[?v_{\mathtt{f}} \setminus v_{\mathtt{f},\mathtt{D}}] \wedge \mathtt{eq}(\mathtt{D})[\mathtt{anchor}_{\mathtt{D}} \setminus v_{\mathtt{f},\mathtt{D}}] \right)$$

*Where all variables $v_{\mathtt{f},\mathtt{D}}$ are fresh, i.e., do not occur anywhere else. Additionally, we get a set of equalities for the form* `this.e` $= x$ *for each backwards edge removed in $\mathfrak{R}(\mathtt{C})$, that maps the field of the source of the backlink to the query variable that is used for the target node of the backwards link.*

*Example 1.* The eager query of class `Bike` in Sec. 2 results in a query that is transformed into the one of Fig. 1 by substituting the nested queries. To illustrate forward-cycle-free classes, consider the following variant of the `Wheel` class.

```
1 class Wheel anchor ?o(
2   Int wheelId, Int last,
3   link(?bike :wheel ?o) Bike bike)
4 end retrieve SELECT ?wheelId ?last WHERE{?o :wheelId ?wheelId; :stamp ?last.}
```

Here, the `Wheel` class has a link to a `Bike` instance. However, analyzing the link clauses, we can see that this is a backwards edge: ?`bike` is always instantiated with the anchor of the outer query when loading `Bike`. Thus, the additional equalities are `this.front.bike` $=?o$ and `this.back.bike` $=?o$.

After constructing the query, it remains to show how we construct an object from a query result. This is done by constructing a store from the variables of the query that are mapped back to their fields. The construction is straightforward, but technically intricate. For the sake of readability, we give it in the technical report [9] and define the signature here.

[1] We assume that we can reorder the cycle so that the potential backwards edge is always as index 1. We remind that the anchor variable is always the first answer variable of a link query, so the concatenation is indeed well-defined.

**Definition 6.** *Let* $\mathtt{C}$ *be a class and* $\mathsf{RS}$ *a set of answers to its eager query. We denote with* $\mathtt{rs2ob}(\mathsf{RS}, \mathtt{C})$ *the objects (cf. Def. 2) created when instantiating* $\mathsf{RS}$*.*

If neither $\mathtt{C}$ nor any class in $\mathfrak{R}(\mathtt{C})$ has a field of list type, then each answer $\mathsf{rs} \in \mathsf{RS}$ corresponds to one instantiated object. To define $\mathtt{rs2ob}(\mathsf{RS}, \mathtt{C})$, we only need to map back from query variables to the field they are associated with and apply the equations generated during query construction for all fields that have no query associated with them. Every class has implicitly a field `String uri` that is instantiated with the URI of the node mapped from the anchor variable. We require that no blank nodes are mapped to anchors.

To connect with the runtime semantics, we add a rule for **load** statements that connect query construction and subsequent object instantiation.

**Definition 7 (Runtime Semantics).** *The runtime semantics of* `MOL` *is the one of Def. 2, extended with the following rule:*

$$\textbf{(eager)}\ \frac{\mathsf{obs}', \mathtt{X} = \mathsf{listOf}(\mathsf{obs}'') \quad \mathsf{obs}'' = \mathtt{rs2ob}(\mathsf{RS}, \mathtt{C}) \quad \mathsf{RS} = \mathsf{ans}(\mathcal{K}, \mathsf{eq}(\mathtt{C}))}{(\mathcal{K}, \mathsf{CT})\mathtt{v := load\ C();\ s}(\sigma, \mathsf{obs}) \to (\mathcal{K}, \mathsf{CT})\mathtt{v := X;\ s}(\sigma, \mathsf{obs}, \mathsf{obs}', \mathsf{obs}'')}$$

The rule executes the query in the first premise. It then creates objects for all results ($\mathsf{obj}''$) and stores them in a list via $\mathsf{listOf}$. The $\mathsf{listOf}$ function returns a pair of objects implementing the list ($\mathsf{obj}'$) and the name of the head object of the list ($\mathtt{X}$). The **load** statement is then reduced to an assignment of this head object to the target variable.

We can ensure that all loaded objects are well-formed, i.e., respect the declared types of their field, by checking whether each query variable respects each declared type. To do so, we check whether the query restricted to this variable is contained in the query retrieving all elements of the declared type, respective its OWL equivalent. The TBox is used to approximate the data statically. The proof follows directly from the typing theorem for semantically lifted programs [4].

**Theorem 1 (Safety).** *Let* $\mathtt{C}$ *be a class. Let* $\varphi = \mathsf{eq}(\mathtt{C})$ *be its eager query and* $V$ *the set of variables within* $\varphi$ *that correspond to data-typed fields. If for each* $v$ *with data type* $D$*, the query containment* $\varphi(v) \sqsubseteq^{\mathcal{T}}_{\mathsf{er}} D$ *holds, then each object created from a result from a KB respecting* $\mathcal{T}$ *of* $\mathsf{eq}(\mathtt{C})$ *is well-formed.*

Finally, we investigate the case where one node occurs in different results and, thus, corresponds to multiple constructed objects. Consider, e.g., the class `class C anchor ?o Int i; retrieve ?o P ?i` and the data set `o1 P 1. o1 P 2.`. This touches a core aspect of the relation between objects, nodes and queries: is an object a container for the results or is it in a bijective relation with some RDF-class? Instead of forcing the developer down one of these roads, we can characterize the situations and provide feedback about the annotated mapping.

**Theorem 2 (Bijective Instantiation for Essentially Functional Classes).** *A forward-cycle-free class* $\mathtt{C}$ *is* essentially functional *in a program* $P$*, if for all paths in* $\mathfrak{R}(P, \mathtt{C})$ *starting in* $\mathtt{C}$*, the concatenation of the labeling queries is functional, and all data properties of reachable classes are functional.*

*In the list retrieved from the retrieval query of an essentially functional class, then there is only one object per node in the answers for the outermost anchor.*

The bijection is established for one execution of one query, not globally. As we are only interested in safe and modular loading, we also do not change the knowledge graph if the instances are manipulated by the program. Similarly, new objects created with **new** are not written into the KB. However, as we will see later, the combination with semantic lifting [10] allows us to write as well.

## 5 Inheritance

We have so far neglected a core element of class-oriented programming: inheritance. Inheritance is, besides the RDFS meta model that defines `rdfs:Class` recursively and uses meta-classes, one of the critical points where OO and OWL class models diverge. Most programming languages forbid, or at least restrict, multiple inheritance, especially diamond inheritance, which causes problems for *methods*. Consequently, in OO, one object cannot be an element of several classes which are not subclasses of each other. It is, thus, out of question to try to reconcile the class model of Java-like languages with the class model of OWL. In this section, we extend our programming language to handle inheritance and give two static analyses that catch modeling errors in the retrieving queries.

**Definition 8 (Syntax of $\texttt{MOL}^+$).** *The syntax of* $\texttt{MOL}^+$ *is based on the syntax of* `MOL`*, with the definition of classes in Def. 3 replaced by the following:*

$$\mathsf{Class} ::= \textbf{class}\ \texttt{C}\ [\textbf{extends}\ \texttt{C}]?\ \textbf{anchor}\ \texttt{X}\ (\overline{[\textbf{link}(\texttt{R})]?\ \texttt{T}\ \texttt{f}})\ \textbf{end retrieve}\ \texttt{Q}$$

The only change of the syntax is the addition of the **extends** clause. Semantically, the only change is the generation of the class table $\mathsf{CT}$, which for any class now also includes the fields of all its superclasses. Thus, $\mathsf{query}(\texttt{C})$ must have the variable for the fields of all its superclasses as well. If `D` has a clause **extends** `C`, then we write $\texttt{D} \leq \texttt{C}$. For the transitive closure, we write $\leq^*$.

*Retrieval Query.* Retrieval for a class that has subclasses must respect these subclasses and construct the most specific class, not necessarily the general one written in the program. For example, consider the following program and ABox.

```
1 class C anchor ?o () end retrieve {?o a :Q}
2 class D anchor ?o extends C (Int j)
3   end retrieve {?o a :Q. ?o :j ?j.}
4 class E anchor ?o extends C (Int k)
5   end retrieve {?o a :Q. ?o :k ?k.}
```

```
obj1 a :Q,  obj1 :j 1,
obj2 a :Q,  obj2 :k 1,
```

When executing **load** `C()`, one would expect that `obj1` is loaded as a `D` instance, because we can retrieve data for the `j` field, and, analogously `obj2` as a `E` instance. Running the `C` query, however does not detect this – it is necessary to adapt our expansion of the retrieval query. Intuitively, we run the queries of the subclasses in `OPTIONAL` clauses and check during object construction whether a given subclass can be instantiated, by checking whether all *variables* belonging to this subclass are instantiated. The idea is to put the additional fields in optional clauses of the query and check whether they are instantiated. If they are, one can downcast the created object.

**Definition 9 (Runtime Semantics of $\texttt{MOL}^+$).** *Let $\texttt{C}$ be a class with subclasses $(\texttt{D}_i)_{i\in I}$. We define the query* $\texttt{inheq}(\texttt{C}) = \texttt{query}(\texttt{C}) \wedge \bigwedge_{i\in I} \texttt{OPTIONAL}(\texttt{inheq}(\texttt{D}_i))$. *The object* $\texttt{rs2ob}^+(\texttt{rs},\texttt{C})$ *retrieved from a result* rs *of* $\texttt{inheq}(\texttt{C})$ *is constructed as follows. Let* $(\texttt{E}_i)_{i\in J} = \{\texttt{E} \leq^* \texttt{C}\}$ *be the set of subclasses of* C*, such that all fields of* $\texttt{E}_i$ *correspond to an assigned variable in* rs*.*

$\texttt{rs2ob}^+(\texttt{rs},\texttt{C}) = \texttt{rs2ob}(\texttt{rs},\texttt{E})$ *for some* $\texttt{E} \in (\texttt{E}_i)_{i\in J}$ *where* $\texttt{rs2ob}(\texttt{rs},\texttt{E})$ *is defined.*

*The runtime semantics of* $\texttt{MOL}^+$ *is the one of* LMOL*, except that every occurrence of* rs2ob *is replaced by* $\texttt{rs2ob}^+$*, and every occurrence of* query *by* inheq*.*

Two unintuitive effects can occur during the retrieval. First, the object instantiation is nondeterministic, and second, it may violate behavioral subtyping.

*Unique Retrieval.* Nondeterminism occurs if two optionals corresponding to unrelated classes are instantiated. E.g., the ABox {`o3 a :Q; :j 1; :k 1.`} used with the above code. Object `o3` can be retrieved as *both* `D` and `E`, but in our class model it is not possible for an object to be both. There are four solutions: (1) Introduce a new language mechanism that allows objects to have the fields of two classes without a subtyping relation between them. (2) Define a preference relation, e.g., say that instantiating `D` is always preferred over instantiating `E`. (3) Retrieve multiple objects, one for each possible instantiation. (4) Take the least specific class of all possible instantiations, i.e., here: `C`.

We deem (1) as unpractical as it changes the programming language according to a specific use case and therefore counteracting our aim of easy to use integration of RDF into OO. Similarly, we deem (2) as unpractical as it questionable if such a preference relation is sensible in many applications. Solution (3) leads to a one-to-many relation between loaded objects and anchored nodes, which we consider undesirable. We, thus, use (4), but give a static analysis that detects the situation where this design decision may play a role: If the conjunction of the queries for each pair of subclass is unsatisfiable, then the most specific constructed class is uniquely determined.

**Theorem 3 (Unique Retrieval).** *Let $\texttt{C}$ be a class with subclasses $(\texttt{D}_i)_{i\in I}$. If for all $\texttt{D}_j, \texttt{D}_k$ with $j,k \in I, j \neq k$ the query* $\texttt{query}(\texttt{D}_j) \wedge \texttt{query}(\texttt{D}_k)$ *is unsatisfiable, then the function* $\texttt{rs2ob}^+(\texttt{rs},\texttt{C})$ *in Def. 9 is deterministic.*

If the check fails, we can precisely give feedback which two clauses overlap and give the programmer detailed feedback where the mapping between RDF and the $\texttt{MOL}^+$ class models fails.

*Behavioral Subtyping.* Given a class `C` with a direct subclass `D`, we must relate their **retrieve** clauses, such that each retrieved object is also a `C` object. This is essentially a case of the Liskov principle [5] of behavioral subtyping: if a property $\varphi(x)$ holds for all instances $x$ of class `C`, then $\varphi(y)$ must also hold for all instances $y$ of all subclasses of `C`. In our case, the properties in question are all data class invariants over the fields of the superclass. For example, consider the above example again, but with the definition of `D` changed to the following:

```
class D anchor ?o extends C (Int j) end retrieve ... {?o a :R. ?o :j ?j.}
```

Here, if `:R` is not a sub-class of `:Q` (in the sense of RDF), then running the query for `D` will retrieve objects that are not retrieved by the query for `C` *even when restricted to the fields of* `C`.

The Liskov principle for $\texttt{MOL}^+$ is reducible to query containment. Given a TBox $\mathcal{T}$, we check, if for all KBs respecting some TBox $\mathcal{T}$ and for all classes, `C`, `D`, with $\texttt{D} \leq \texttt{C}$, the query of the subclass does not add new instances when restricted to the fields of the superclass.

**Theorem 4 (Behavioral Subtyping for $\texttt{MOL}^+$).** *Let* $\texttt{C}, \texttt{D}$ *be two classes* $\texttt{D} \leq \texttt{C}$ *and* $\bar{f}_{\texttt{C}}$ *is the set of fields in the superclass. If for each such pair of classes the query containment* $\mathsf{eq}(\texttt{C})(\bar{f}_{\texttt{C}}) \sqsubseteq^{\mathcal{T}} \mathsf{eq}(\texttt{D})(\bar{f}_{\texttt{C}})$ *holds, then*

$$\left\{\mathtt{rs2ob}(\mathtt{rs}, \texttt{C}) \mid \mathtt{rs} \in \mathsf{ans}(\mathcal{K}, \mathsf{eq}(\texttt{C}))\right\} \subseteq \left\{\mathtt{rs2ob}(\mathtt{rs}, \texttt{C}) \mid \mathtt{rs} \in \mathsf{ans}(\mathcal{K}, \mathsf{eq}(\texttt{D}))\right\}$$

## 6 Lazy Loading

We now extend $\texttt{MOL}^+$ to `LMOL` by introducing a lazy loading mechanism. Lazy loading splits the eager query into several subqueries, of which some are delayed and only executed on demand, as has advantages for usability and performance.

First, it gives the programmer very precise control over the used data. Instead of loading all possible data that *may* be used, it enables to load data as it is indeed used. In our running example, it may depend on the data loaded for the `Bike` instance whether the `front` or `back` wheel must be investigated. This condition may not be (easily) encodable in the query, or indeed not be known upfront and depend on user input or data loaded from other sources. For example, the following program accesses three bikes, but only *two* wheels: the front wheel of the second result and the back wheel of the third one.

```
List<Bike> l := load Bike(); l[0].id; l[1].front.id; l[2].back.id
```

It is easy to see how in a more complex language one can decide which wheel to access based on prior data. Lazy loading can thus solve the problem of loading data that is not required in the application.

Second, lazy loading decouples modeling the data mapping from writing the query for a specific optimization: the programmer can be more generous in data modeling, as the specialization to a specific computation occurs at runtime.

Syntactically, we add futures to the types and a statement to resolve them.

**Definition 10 (Syntax of `LMOL`).** *The syntax of* `LMOL` *is the syntax of* $\texttt{MOL}^+$ *of Def. 8, with the following extensions for types and right-hand-side expressions:*

$$\texttt{T} ::= \ldots \mid \texttt{QFut<C>} \qquad \texttt{rhs} ::= \ldots \mid \textbf{load}\ \texttt{e}$$

Intuitively, a future is a delayed expression, in our case a query. We use explicit futures here [11] that must be explicitly resolved. For resolving, we reuse the **load** keyword: a **load** `e` expression takes an expression `e` of `QFut<C>` type and

returns an expression of `List<C>` type – the results of executing the delayed query.[2] A future field cannot be a backwards edge.

Next, we augment the runtime with the required elements for futures. *Anchor maps* keep track of the variable instantiations of the previously executed queries.

**Definition 11 (Runtime Configurations of LMOL and Lazy Queries).** *Runtime configurations are extended as follows. Let* `F` *be the set of future identifiers, a subset of the object identifiers, and* `A` *the set of anchor map identifiers, which is disjoint to the set of object identifiers. Let* `Q` *range over SPARQL queries and object identifiers.*

$$\mathsf{conf} ::= (\mathcal{K}, \mathsf{CT})\ \mathtt{s}\ (\sigma, \mathsf{obs}?, \mathsf{futs}?, \mathsf{acs}?)$$
$$\mathsf{futs} ::= \mathbf{fut}(\mathtt{F}, \mathtt{A}, \mathtt{Q}) \mid \mathsf{futs}\ \mathsf{futs} \qquad \mathsf{acs} ::= \mathbf{ac}(\mathtt{A}, \mathcal{A}) \mid \mathsf{acs}\ \mathsf{acs}$$

*where* $\mathcal{A}$ *are maps from RDF literals to object identifiers. Additionally,* $\sigma$ *and all stores* $\rho$ *may map to future identifiers* `F`*. The lazy query of a class is defined analogous to its eager query, except that none of the queries of the fields with future type are executed. Only the linking queries are executed.*

$$\mathtt{lq}(\mathtt{C}) = \mathtt{query}_{\mathtt{C}} \wedge \bigwedge_{(\mathtt{f},\mathtt{D}) \in \mathtt{cf}(\mathtt{C})} \left( \mathtt{link}_{\mathtt{C},\mathtt{f}}[?v_{\mathtt{f}} \setminus v_{\mathtt{f},\mathtt{D}}] \right)$$

Object instantiation is analogous and described in the technical report. The main differences are that (1) the fields of future type are instantiated with runtime futures, whose query is the query of the class enhanced with the link query where the anchor variable is replaced by its instantiation, and that (2) an anchor map is used as an additional parameter. The anchor map keeps track of *all* instantiations so far. Instantiation for lazy class loading $\mathtt{lrs2ob}(\mathsf{RS}, \mathtt{c})$ thus takes an answer set $\mathsf{RS}$ and the class `c` as input and returns a set of objects, a set of futures and an anchor map. Lazy instantiation for the inner queries, $\mathtt{llrs2ob}(\mathsf{RS}, \mathtt{c}, \mathcal{A})$ takes additionally an anchor map as input. It returns a set of objects and a set of futures, as well as a modified anchor map with added bindings.

**Definition 12 (Runtime Semantics of LMOL).** *The rule for* **load** `C()` *is almost the same as* **(eager)**, *except that we use* $\mathsf{lq}$ *instead of* $\mathsf{eq}$*. It is given in the technical report. The rule for* **load** `e` *is as follows:*

$$\textbf{(lazye)}\ \frac{[\![\mathtt{e}]\!]^{\sigma}_{\mathsf{obs}} = \mathtt{F} \quad \mathsf{RS} = \mathsf{ans}(\mathcal{K}, \mathtt{Q}) \quad \mathsf{obs}', \mathtt{X} = \mathsf{listOf}(\mathsf{obs}'') \quad \mathsf{obs}'', \mathsf{futs}', \mathsf{acs}' = \mathtt{llrs2ob}(\mathsf{RS}, \mathtt{c}, \mathcal{A})}{\begin{array}{l}(\mathcal{K}, \mathsf{CT})\mathtt{v := load\ e;\ s}(\sigma, \mathsf{obs}, \mathsf{futs}, \mathbf{fut}(\mathtt{F}, \mathtt{A}, \mathtt{Q}), \mathsf{acs}, \mathbf{ac}(\mathtt{A}, \mathcal{A})) \\ \quad \rightarrow (\mathcal{K}, \mathsf{CT})\mathtt{v := X;\ s}(\sigma, \mathsf{obs}, \mathsf{obs}', \mathsf{obs}'', \mathsf{futs}, \mathsf{futs}', \mathsf{acs}, \mathsf{acs}')\end{array}}$$

The rules work analogously: First, the lazy query (either by constructing it for the class, or by reading it from the future) is executed. Then objects and futures are instantiated for its results and the anchor map is updated. Finally, the objects are stored in a list and all created constructs are added to the state.

[2] We refrain from introducing (a) expressions for resolved futures and (b) lazy loading of the result list. Both is standard and orthogonal to lazy loading *within* one result.

To be clear, we do not save the result when resolving a future, so a future might be resolved multiple times. The mechanism to avoid this is straightforward [6] and would only obfuscate our contribution.

**Theorem 5.** *For forward-cycle-free classes* `C`*, such that every class in* $\mathfrak{R}(P, \texttt{C})$ *has only data-type and list-type fields,* $\texttt{MOL}^+$ *and* `LMOL` *load the same results, i.e., if all futures are resolved, then the objects in the list of the first* **load** `C()` *contain the same elements, except with one* **fut** *reference in every list field.*

For general classes, `LMOL` can load more or less data. As an example for more data, consider that futures can be used to encode streams [12], and thereby load an unbounded number of objects within *one* result of the overall query. For less data, consider a knowledge base where the bike instances have no stored wheels. Executing the eager query will return no bikes, but executing the lazy query will return all the bikes, with empty lists for their wheels.

## 7 Evaluation

The on-going implementation is available as open-source software. `LMOL` is implemented as an interpreter that takes a `LMOL` file, and a RDF file for external data as input. The interpreter, including the experiments described below, is available under `https://github.com/Edkamb/SemanticObjects/tree/lazy`.

To assess the runtime overhead and memory consumption caused by delaying parts of the query through futures, which reduce the possibilities of the DBMS for query optimization, we run the following experiment. We generate $n$ classes of the form **class** $\texttt{C}_i$`(Int` $\texttt{f}_i$`,` $\texttt{C}_{i+1}$ `next)` **end** with the obvious link and retrieval query, as well as a lazy version of the same form with fields `QFut<`$\texttt{C}_{i+1}$`>` `next`. We consider two scenarios: scenario 1 uses a dataset with two loadable nodes $a_i$, $b_i$ of each class $\texttt{C}_i$, with links from $a_i$ to $a_{i+1}$ and $b_i$ to $b_{i+1}$, making two paths of length $n$. Scenario 2 has additional links from $a_i$ to $b_{i+1}$ and $b_i$ to $a_{i+1}$, leading to $2^n$ possible paths. We evaluate how fast it is to load every possible node into the OO structure and compare three runs for each scenario: one with the eager query, and two with the lazy query, which uses the delay to remove duplicate objects based on their URI before running the next query: The first lazy run accesses only the first class, the second accesses the last class. Fig. 3 shows the results. As expected, there is no performance gain for the simple scenario, but a considerable one for the one with more interconnected data, especially when only parts of the data is accessed. For a class chain of $n = 18$, eager evaluation requires 18s, while lazy evaluation with fine-grained control requires 2.6s if only half of the chain is accessed ($-85\%$). For higher $n$, eager evaluation runs out of heap space. Memory consumption behaves, as expected, like the time consumption.

To confirm that our approach indeed simplifies the design of realistic queries, we remodeled the queries used to access the Slegge database of Equinor of subsurface exploration data [13]. The aim of this is to show that our system enables reuse and forward-cycle-free classes are not a strong restriction. We remodeled

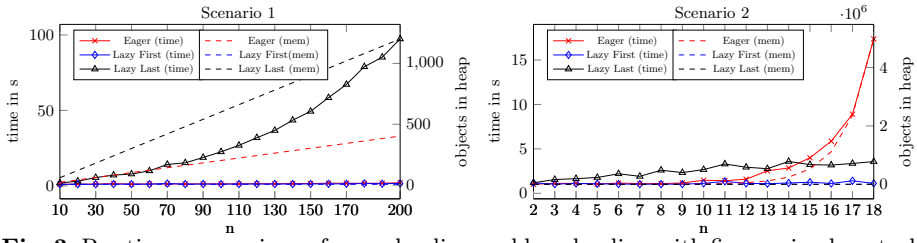



**Fig. 3.** Runtime comparison of eager loading and lazy loading with fine-grained control.

the 8 SPARQL queries for the main, first information need of Slegge need using `LMOL` in a class structure of 21 classes, each corresponding to a reusable pattern or original query. The code below is a representative excerpt. The eager queries for `WBQ1` and `WBQ3` correspond to one of the original Slegge queries, which all retrieve wellbores based on different criteria. `WBQ1` and `WBQ3` only differ in restrictions on the interval, encoded in different classes (`SZ13` and `DepthSZ`). All other parts of the queries, e.g. the wellbore name, are shared through common superclasses.

```
1 class WBWithName extends WB anchor ?w ( String name )
2  end retrieve "?w a :Wellbore. ?w :name ?name."
3 class WBQ1 extends WBWithName anchor ?w (
4  link("?w :wellboreInterval ?int.") SZ13 int, ..) end ..
5 class WBQ3 extends WBWithName anchor ?w (
6  link("?w :wellboreInterval ?int.") DepthSZ int, ..) end ..
```

## 8 Related Work

`LMOL` is implemented on top of the semantic lifting language `SMOL`. Semantic lifting [10] also integrates OO and RDF: It exports a program state and allows to query it then, thus realizing data writes through state change. Consequently, an RDF graph can be changed in any way using `LMOL`/`SMOL`.

`LMOL` is the first language with modular queries and lazy evaluation for RDF data. In the following, we discuss other approaches that connect OO programming and RDF data. Frameworks like Apache Jena [14] or RDF4J [15] connect OO programming and RDF as well, but do not connect the object *models*.

The impedance mismatch for relational databases has been extensively studied for several decades [1] and we refer to Ireland [16] for a comprehensive discussion. It is worth noting that one of the systems to connect OO with relational databases, the LINQ [17] framework for .Net, has been extended to RDF [18]. LINQ provides its own query language, which is mapped to different storage endpoints, and provides no safety mechanisms or lazy evaluation. The query is not modular and provided at the loading statement. For RDF, impedance mismatch has been explored, starting with Goldman [19] and the Go! language [20]. An in-depth discussion is given in the survey of Baset and Stoffel [3].

The most common approach taken is to relate OWL concepts to OO classes directly. For example, Leinberger et al. [21] use a special query language to load

RDF data into an OO language by relating OWL concepts to OO types. Their query language is typable and translates into SPARQL. In contrast, `LMOL` supports full SPARQL and does not require the user to learn an additional query language. Owl2Java [22], Agogo [23] or ActiveRDF [24] are similar approaches, suffering from the impedance mismatch. They all generate OO classes based on some RDF schema for a certain target language and establish a direct connection this way, where only certain RDF schemata are allowed. In contrast, we give a way to detect whether the defined mapping establishes a one-to-one correspondence in the results of one query (Thm. 2) to help the programmer.

While our approach embraces the semantic gap between OO and RDF object models, and the approaches so far attempt to bridge it, Eisenberg and Kanza [2] attempt a unification in a programming model that treats RDF individuals as program primitives. This essentially imports the RDF object model into the programming language, and the authors do not discuss typing. Indeed, they present their approach for Ruby, with a loose object model using dynamic duck typing.

As for type checking, Seifer et al. [25] give a system where DL concepts are types and that requires to type check the SPARQL query itself. In contrast, the entailment-based system we use is more modular: we do not have any restrictions on the used SPARQL subset. Furthermore, we do not entangle type checking in the impedance mismatch by mixing concepts and types.

Leinberger [26] gives an extended type system based on SHACL and shape containment, instead of SPARQL and query containment. That approach focuses on ensuring the existence of data that is loaded and is neither modular nor supports lazy evaluation, as the semantic gap is only considered at the interface.

## 9 Conclusion

We have presented a connection between RDF and OO programming that maintains and embraces the semantic gap between the underlying inheritance mechanisms and object models, and allows modular data modeling, data access and type safety checks. Our approach is the first to explore lazy evaluation of SPARQL queries within a single answer and the first to formally define a Liskov principle for a connection of OO and RDF. Our evaluation shows that lazy evaluation leads to significant performance gains for loading of complex data.

We plan to generalize our prototype to a static tool for a mainstream programming language and increase its expressive power by (1) using a `Option<C>` type that maps to optionals in the retrieval query and (2) allowing the **load** statement to take an additional parameter that defines additional constraints.

**Acknowledgments** This work was supported by the RCN via `PeTWIN` (294600). The authors thank Dirk Walther for motivating this work and the anonymous reviewers for the constructive feedback.

## References


1. George P. Copeland and David Maier. Making smalltalk a database system. In Beatrice Yormark, editor, *SIGMOD*, pages 316–325. ACM Press, 1984.
2. Vadim Eisenberg and Yaron Kanza. Ruby on semantic web. In *ICDE*, pages 1324–1327. IEEE Computer Society, 2011.
3. Selena Baset and Kilian Stoffel. Object-oriented modeling with ontologies around: A survey of existing approaches. *Int. J. Softw. Eng. Knowl. Eng.*, 28(11-12):1775–1794, 2018.
4. Eduard Kamburjan and Egor V. Kostylev. Type checking semantically lifted programs via query containment under entailment regimes. In *Description Logics*, volume 2954 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021.
5. Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
6. Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
7. Henry G. Baker and Carl Hewitt. The incremental garbage collection of processes. In James Low, editor, *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, USA, August 15-17, 1977*, pages 55–59. ACM, 1977.
8. Gordon Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61, 2004.
9. Eduard Kamburjan, Vidar Norstein Klungre, and Martin Giese. Never mind the semantic gap: Modular, lazy and safe loading of rdf data (technical report). Research report 502, Dept. of Informatics, University of Oslo, March 2022.
10. Eduard Kamburjan, Vidar Norstein Klungre, Rudolf Schlatte, Einar Broch Johnsen, and Martin Giese. Programming and debugging with semantically lifted states. In *ESWC*, volume 12731 of *LNCS*, pages 126–142. Springer, 2021.
11. Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, 2017.
12. Keyvan Azadbakht, Frank S. de Boer, Nikolaos Bezirgiannis, and Erik P. de Vink. A formal actor-based model for streaming the future. *Sci. Comput. Program.*, 186, 2020.
13. Dag Hovland, Roman Kontchakov, Martin G. Skjæveland, Arild Waaler, and Michael Zakharyaschev. Ontology-based data access to slegge. In *ISWC (2)*, volume 10588 of *LNCS*, pages 120–129. Springer, 2017.
14. Apache Foundation. Apache jena. `https://jena.apache.org/`.
15. Eclipse Foundation. Eclipse RDF4J. `https://rdf4j.org/`.
16. Jon Christopher Ireland. *Object-relational impedance mismatch : a framework based approach*. PhD thesis, Open University, Milton Keynes, UK, 2011.
17. Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .net framework. In *SIGMOD*, page 706. ACM, 2006.
18. Andrew Matthew. LINQtoRDF, 2006. https://code.google.com/archive/p/linqtordf/.
19. Neil M. Goldman. Ontology-oriented programming: Static typing for the inconsistent programmer. In *ISWC*, volume 2870 of *LNCS*, pages 850–865. Springer, 2003.
20. Keith L. Clark and Frank G. McCabe. Ontology oriented programming in go! *Appl. Intell.*, 24(3):189–204, 2006.

21. Martin Leinberger, Stefan Scheglmann, Ralf Lämmel, Steffen Staab, Matthias Thimm, and Evelyne Viegas. Semantic web application development with LITEQ. In *ISWC*, volume 8797 of *LNCS*, pages 212–227. Springer, 2014.
22. Aditya Kalyanpur, Daniel Jiménez Pastor, Steve Battle, and Julian A. Padget. Automatic mapping of OWL ontologies into java. In *SEKE*, pages 98–103, 2004.
23. Fernando Silva Parreiras, Carsten Saathoff, Tobias Walter, Thomas Franz, and Steffen Staab. APIs à gogo: Automatic generation of ontology APIs. In *ICSC*, pages 342–348. IEEE Computer Society, 2009.
24. Eyal Oren, Benjamin Heitmann, and Stefan Decker. ActiveRDF: Embedding semantic web data into object-oriented languages. *J. Web Semant.*, 6(3):191–202, 2008.
25. Philipp Seifer, Martin Leinberger, Ralf Lämmel, and Steffen Staab. Semantic query integration with reason. *Art Sci. Eng. Program.*, 3(3):13, 2019.
26. Martin Leinberger. *Type-safe Programming for the Semantic Web*. PhD thesis, University of Koblenz and Landau, Germany, 2021.