Semantic Query Integration With Reason

Philipp Seifer[a], Martin Leinberger[b], Ralf Lämmel[a], and Steffen Staab[b,c]

a Software Languages Team, University of Koblenz-Landau, Germany

b Institute for Web Science and Technologies, University of Koblenz-Landau, Germany

c Web and Internet Science Research Group, University of Southampton, England

Abstract Graph-based data models allow for flexible data representation. In particular, semantic data based on RDF and OWL fuels use cases ranging from general knowledge graphs to domain specific knowledge in various technological or scientific domains. The flexibility of such approaches, however, makes programming with semantic data tedious and error-prone. In particular the logics-based data descriptions employed by OWL are problematic for existing error-detecting techniques, such as type systems. In this paper, we present DOTSpa, an advanced integration of semantic data into programming. We embed description logics, the logical foundations of OWL, into the type checking process of a statically typed programming language and provide typed data access through an embedding of the query language SPARQL. In addition, we demonstrate a concrete implementation of the approach, by extending the Scala programming language. We evaluate the integration by comparing programs using our approach to equivalent programs using a state of the art library in several dimensions—difficulty and effort using the Halstead metric, compilation and runtime performance, as well as the size of compiled artifacts.

ACM CCS 2012

- *Theory of computation → Type structures;*
- *Software and its engineering → Syntax; Semantics; Compilers;*

Keywords DSL implementation, compiler extension, type checking, language Integration

The Art, Science, and Engineering of Programming

Perspective The Art of Programming

Area of Submission General-purpose programming, Program verification, Semantic data

1 Introduction

Graph-based data models allow for flexible data representation. In particular, semantic data models like RDF [73] may contain schematic information as part of the data or in separate files. Such schematic information is called an ontology. Of special interest is the W3C standard OWL [39], which allows for using highly expressive logic-based data descriptions. This flexibility and expressive power of RFD and OWL fuels many applications, ranging from general knowledge graphs such as Wikidata [72] to complex domain specific ontologies, e.g., the Snomed CT [10] medical vocabulary.

While the flexibility and expressive power of OWL make it attractive, programming with OWL is tedious and error-prone. A major reason is the lack of typed integration in programming languages, leaving the burden of correct typing on the programmer. This lack of integration is comparable to data access and integration of other data models, such as access and types for relational [9] or object oriented databases [53, 74], XML [7, 40], as well as general data access approaches such as LINQ [8, 50]; each data model comes with its specific challenges. As an example of these challenges, consider the following axioms inspired by the Lehigh University Benchmark [31]:

```
1  // Schematic information
2  Person ⊓ Organization ⊑ ⊥
3  Employee ⊑
4      Person ⊓ ∃worksFor.Organization
5  Professor ⊑ Employee
6  Chair ⊑ Professor
7  ∃headOf.Department ⊓ Person ≡ Chair
8  ResearchAssistant ⊑
9      Person ⊓ ∃worksFor.ResearchGroup
10 Department ⊑ Organization
11 ResearchGroup ⊑ Organization
12 ∃worksFor.⊤ ⊑ Person
13 ⊤ ⊑ ∀headOf.Department
14
15 // Data / assertional axioms
16 alice : Chair
17 (bob, softlang) : worksFor
18 softlang : ResearchGroup
```

Figure 1 Example axioms describing a university setting.

Schematic information consists of concepts, such as *Person*, and roles, such as *worksFor*, that are combined to more complex concept expressions via connectors such as intersection (⊓) or existential (∃) and universal (∀) quantification. Concepts themselves are related to each other via subsumption (⊑) or equivalence (≡) statements. In this manner, line 2 ensures that a *Person* can never be an *Organization* and vice versa. Lines 3–4 state that *Employees* are *Persons* that work for some kind of *Organization*. Line 5 introduces *Professors*, which are *Employees*. A special kind of a *Professor* is a *Chair* (line 6) which is a person who is the head of a department (line 7). A *ResearchAssistant* is a person that works for a *ResearchGroup* (line 8–9). Both *Departments* and *ResearchGroups* are special kinds of *Organizations* (lines 10 and 11). Line 12 acts as a domain specification, ensuring that everyone who works for something is considered a person. Line 13 constitutes a range specification, ensuring all objects to which a *headOf* relation points are *Departments*. The data introduces three objects—*alice*, who is a *Chair*, and *bob* who works for the *ResearchGroup softlang*.

This example highlights some of the problems that occur when trying to do type checking on a program that works on OWL. For one, a mixture of nominal (*Person*) and structural types (∃*worksFor*.*Organization*) is used. Second, some information is left implicit—such as the fact that a *ResearchAssistant* is a special kind of *Employee*.

Mapping approaches such as [41] do not cope well with these problems. In previous work, we proposed a custom type system dubbed λ_{DL} to remedy the situation [45]. λ_{DL} used concept expressions such as $\exists$*worksFor*.*ResearchGroup* or *Employee* as types. The process of type checking then relies on an ontology reasoner. This allows for the definition of functions, e.g., a function accepting an *Employee* such as (λx : *Employee* . ...), as well as proving that a *ResearchAssistant* is a subtype of *Employee* through the reasoner. Besides the possibility of finding wrong applications of this function at compile time, this also serves as documentation, which is guaranteed to be consistent [58]—with both the program code as well as the ontology.

A practical integration of OWL into programming, however, must extend general purpose programming languages. Those have rich type systems, where even small changes may be cross-cutting among many features. In addition, expressive queries as a means to access data are needed. In particular, a typed integration of SPARQL [59], a W3C querying standard for semantic data, is desirable.

In this paper, we describe a general approach for a deep integration of OWL and a subset of SPARQL into a typed programming language as well as a concrete implementation, ScaSpa, as an extension of Scala. The subset of SPARQL we consider is built on [42] in order to focus on SPARQL constructs that are decidable when used with OWL. In summary, the **main contributions** of the paper are as follows:

1. We devise an advanced approach for the integration of semantic data into programming. This includes on-demand type integration (we only rely on concepts used in the program) based on the theoretic foundations provided by λ_{DL}, as well as a deep integration of concept expressions and SPARQL queries. This allows for the detection of three kinds of common errors, that occur when working with OWL.
2. We provide a concrete implementation of this approach by extension of the Scala language, including type erasure, integration of an existing reasoner and triple store, while maintaining separate compilability.
3. We show the reduced complexity that arises from this integration using a metrics-based evaluation by comparing programs written with ScaSpa to programs using a traditional approach.

However, two important issues are not addressed by the paper. For one, we currently do not provide any form of code completion or general design support dedicated to SPARQL and DL concept expressions. Second, we do not conduct user studies to verify that the reduced complexity is relevant in practice. Both of theses issues call for future work.

Road Map In Section 2, we introduce description logics and SPARQL. In Section 3, we show an essential part of the integration by inferring query types from SPARQL queries. Section 4 describes the essence of integrating DL and SPARQL with typed functional object oriented programming, by extending the syntax and semantics of a formal calculus. In Section 5, we discuss several issues regarding the practical integration of description logics and SPARQL. We also illustrate the approach through a small example using our implementation. Section 6 gives an overview of the architecture and implementation of ScaSpa. In Section 7, we evaluate the effects of our approach

onto effort and difficulty of programs. Our empirical analysis uses our Scala-based implementation as an instance of the described approach, for comparing API code versus type-checked, language-integrated DL and SPARQL code. This is followed by a discussion of related work in Section 8 and a short summary in Section 9.

2 Background

In this paper, we focus on semantic data formalized in the Web Ontology Language (OWL). Formal theories about OWL are grounded in research on description logics (DL).[1] Description logics are a family of logical languages used in knowledge representation. They are sub-languages of first-order logic, being defined to allow for decidable or even PTIME decision procedures.

Description Logics A DL knowledge base $\mathcal{K}$ typically comprises two sets of logical axioms: The T-Box (terminological or schematic data) and the A-Box (assertional data). Such axioms are built using the atomic elements defined in the signature of $\mathcal{K}$. The signature provides a set of atomic concept names (e.g., *Person* or *ResearchGroup*), set of atomic role names (e.g., *worksFor*) and atomic object names (e.g., *bob* or *softlang*).

A role expression R is either an atomic relation or its inverse. Atomic concept names, role expressions and individual objects can be used to built complex concept expressions C using connectives. The available connectives depend on the specific dialect of the description logic. Common connectives include conjunction ($\sqcap$), negation ($\neg$), existential quantification ($\exists$) and enumeration of objects for concept creation. Other connectives (disjunction, universal quantification, ...) may be derived from these. Semantically, a concept is a set of objects. T-Box axioms are constructed from a pair of concept expressions using either the subsumption connective ($\sqsubseteq$) or the equivalence connective ($\equiv$), essentially describing subsumption or equivalences between the various sets of objects. For example, the axiom $ResearchGroup \sqsubseteq Organization$ describes that all objects contained in the set *ResearchGroup* must also be contained in the *Organization* set. Assertional axioms on the other hand are either concept assertions or role assertions. A concept assertion $a : C$ claims membership of an object a in a concept expression C (e.g., $softlang : ResearchGroup$ meaning that the *softlang* object is contained in the *ResearchGroup* set). A role assertion $(a, b) : R$ (e.g., $(bob, softlang) : worksFor$) connects objects via role expressions.

Being a subset of first-order predicate logic, DL relies on a Tarski-style *interpretation based semantics*. Axioms contained in either the T-Box or A-Box of $\mathcal{K}$ constitute known facts that must be true in all sensible interpretations of $\mathcal{K}$—such a sensible interpretation is called a *model*. This may introduce anonymous objects. Consider *alice*, who is a *Chair*. Being a *Chair* requires being the head of a department. However, no

[1] In practice, OWL is often serialized using RDF. The strict subject-predicate-object triple style of representation introduces some syntactic differences compared to the abstract syntax introduced in this paper.

department is given for *alice*. This is no inconsistency, but rather incomplete knowledge. An anonymous object is being used in the models of $\mathcal{K}$ to represent this department.

An axiom A can be inferred from a knowledge base, written $\mathcal{K} \models A$ if it is true in every model—for example, $\mathcal{K} \models \mathit{ResearchAssistant} \sqsubseteq \mathit{Employee}$ for the $\mathcal{K}$ given in Figure 1. Importantly, DL relies on an open world assumption. An axiom is true if it is true in all models. It is false, if it is false in all models. If some models exist in which an axiom is true and some where it is false, then we cannot say whether the axiom is true or false for $\mathcal{K}$. Also, DL does not employ a unique name assumption—different object names are considered syntactic elements that may semantically refer to the same thing unless explicitly stated otherwise.

SPARQL in a DL Context SPARQL [59] is a graph-matching language built around query patterns. SPARQL supports various entailment regimes, including OWL entailment [28]. While our implementation relies on the official SPARQL syntax, in the theoretical parts of the paper we rely on an algebraic formalization for simplicity. We follow [42] in our definitions to focus on constructs that are decidable when used in a DL context. In particular, we restrict ourselves to queries to the A-Box.

A query pattern is an axiom in which at least one object is replaced with a variable. We indicate variables through the meta-variables x, x_1, x_2. Therefore, a query pattern *pattern* is defined as follows:

$$\mathit{pattern} ::= x : C \mid (a, x) : R \mid (x, b) : R \mid (x_1, x_2) : R$$

A SPARQL query q is then either a query pattern or the connection of two queries via conjunction, union, minus or optional:

$$\begin{array}{rll} q ::= & \mathit{pattern} & \text{(query pattern)} \\ & \mid q_1 \text{ AND } q_2 & \text{(conjunction)} \\ & \mid q_1 \text{ UNION } q_2 & \text{(union)} \\ & \mid q_1 \text{ MINUS } q_2 & \text{(minus)} \\ & \mid q_1 \text{ OPT } q_2 & \text{(optional)} \end{array}$$

Formally, a possible solution to a query q is a mapping μ from variables used in the query onto objects used in $\mathcal{K}$. We write $\mu(q)$ to denote the result from replacing each variable in q with $\mu(x)$. We write $\mathcal{K} \models \mu(q)$ to indicate that μ is a solution to q. A solution to q is a mapping μ such that:

$$\begin{array}{lll} \mathcal{K} \models a : C & \text{iff} & \mu(q) = a : C \\ \mathcal{K} \models (a, b) : R & \text{iff} & \mu(q) = (a, b) : R \\ \mathcal{K} \models \mu(q_1) \text{ and } \mathcal{K} \models \mu(q_2) & \text{iff} & \mu(q) = \mu(q_1) \text{ AND } \mu(q_2) \\ \mathcal{K} \models \mu(q_1) \text{ or } \mathcal{K} \models \mu(q_2) & \text{iff} & \mu(q) = \mu(q_1) \text{ UNION } \mu(q_2) \\ \mathcal{K} \models \mu(q_1) \text{ and } \mathcal{K} \not\models \mu(q_2) & \text{iff} & \mu(q) = \mu(q_1) \text{ MINUS } \mu(q_2) \\ \mathcal{K} \models \mu(q_1 \text{ UNION } (q_1 \text{ AND } q_2)) & \text{iff} & \mu(q) = \mu(q_1) \text{ OPT } \mu(q_2) \end{array}$$

The answer to a query q for a knowledge base $\mathcal{K}$, written $[\![q]\!]_{\mathcal{K}}$ is the set of all solution mappings μ for which $\mathcal{K}$ entails the query:

$$[\![q]\!]_{\mathcal{K}} = \{\mu | \mathcal{K} \models \mu(q)\}$$

3 Type Inference for SPARQL Queries

$$(x : C) : \phi \text{ with } \phi(x) = C \qquad ((x, a) : R) : \phi \text{ with } \phi(x) = \exists R.\{a\}$$

$$((a, x) : R) : \phi \text{ with } \phi(x) = \exists R^{-}.\{a\}$$

$$((x_1, x_2) : R) : \phi \text{ with } \phi(x_1) = \exists R.x_2 \text{ and } \phi(x_2) = \exists R^{-}.x_1 \qquad \frac{q_1 : \phi_1 \qquad q_2 : \phi_2}{q_1 \text{ AND } q_2 : \phi_1 \odot \phi_2}$$

$$\frac{q_1 : \phi_1 \qquad q_2 : \phi_2}{q_1 \text{ UNION } q_2 : \phi_1 \oplus \phi_2} \qquad \frac{q_1 : \phi_1 \qquad q_2 : \phi_2}{q_1 \text{ MINUS } q_2 : \phi_1}$$

where for $(\mathbf{o}, \mathbf{so}) \in \{(\odot, \sqcap), (\oplus, \sqcup)\}$
$\phi_1 \ \mathbf{o} \ \phi_2 = \{(x, \phi_1(x) \ \mathbf{so} \ \phi_2(x)) \mid x \in dom(\phi_1), x \in dom(\phi_2)\}$
$\cup \{(x, \phi_1(x)) \mid x \in dom(\phi_1), x \notin dom(\phi_2)\}$
$\cup \{(x, \phi_2(x)) \mid x \notin \phi_1, x \in \phi_2\}$

Figure 2 Rules for concept inference on queries.

In order to provide a typed integration of SPARQL queries into programming, type inference on queries is needed. From a semantics' point of view, a concept expression is a set of values. Queries evaluate to sets of mappings that map variables to values. We therefore infer one concept expression per variable of a SPARQL query. The set defined through the concept expression must at least contain all possible values that a variable can be mapped to after the query has been evaluated. We define the type of a query to be a function ϕ mapping each variable in the query to a concept expression.

We use a static analysis of the query through a typing relation $q : \phi$. Query patterns constitute the basic cases of this analysis. In case of a $x : C$ pattern, all possible mappings for x are members of concept C. Likewise, for $(a, x) : R$ and $(x, a) : R$, all possible mappings must belong to $\exists R^{-}.\{a\}$ and $\exists R.\{a\}$ respectively. We use $\{a\}$ to denote a so called nominal concept—a concept created by enumerating all its objects. A special case is $(x_1, x_2) : R$. As the concrete concept for x_1 is dependent on the concept for x_2 and vice versa, we introduce concept references $\exists R.x$ for each of the two variables. These references get resolved after the query has been analyzed. Conjunction and disjunction in queries are transformed into conjunctions or disjunctions of DL concept expressions in cases where variables are contained in both parts of the query. For *MINUS* queries we have to overestimate the types by disregarding all constraints of the right-hand side. The *MINUS* operator in SPARQL evaluates both operands, before removing all left-hand side solutions incompatible with the right-hand side. Therefore, the overestimation is sound (but a superset of the precise type). We could not express this more accurately using concept negation, however, since this notion of negation differs from SPARQL. The full rules are shown in Figure 2.

Concept references are resolved in the last step. A concept reference $\exists R.x$ is substituted with the respective concept in ϕ, yielding $\exists R.\phi(x)$. This is repeated until all concept references are eliminated except possible self references. These cases take the form $\phi(x_1) = \exists R.x_1$ or similar. As we need to replace the reference in a way that captures all possible values, we replace it through the $\top$ concept, yielding $\phi(x_1) = \exists R.\top$. As $\top$ represents the concept containing all objects, this may be a very loose, but sound overestimation.

4 Syntax and Semantics of DOTSpa

In previous work, we introduced description logics based types to a simply typed lambda calculus [45]. Here we present syntax and semantics of DOTSpa as extensions to an unspecified formalism. We therefore abstract from specific details, in order to keep DOTSpa as general as possible. In the context of the ScaSpa implementation, however, these definitions can be understood as extensions to the dependent object types calculus (DOT [3]), the theoretical foundation of Scala. In fact, the syntax is a direct extension of the calculus, while the reduction, type assignment and subtyping rules are generalized from the object based nature of DOT.

Syntax The syntax extension defined by DOTSpa is given in Figure 3. It extends the rules for values, terms and types. Simple values include literals for internationalized resource identifiers (IRIs), which – consistent with SPARQL – are used to refer to A-Box instances. Terms are extended by adding the various terms defined by DOTSpa: SPARQL queries and their strictly validated variant, role projections and type case expressions. Strict SPARQL queries use a different validation mechanism than non-strict queries, but are otherwise identical. Role projections query along a single role, providing an easy shorthand notation for this common operation. Type cases are branching expressions, which select one of their branches based on subtyping: They consist of a term on which cases are matched, the default case and an arbitrary number of additional cases.

Types can now be expressed by concept expressions to form concept expression types, using common DL syntax. Additionally, nominal and atomic concepts as well as atomic roles are expressed by IRIs. The remaining rules specify our simplified SPARQL queries (as introduced in Section 2). In this version, however, queries might also contain arbitrary terms in addition to SPARQL variables. This allows the embedding of terms from the language context within a query.

Semantics Figure 4 sketches the reduction rules for DOTSpa. Similar to the syntax extension, we specify only rules unique to DOTSpa and omit rules for simple term reduction.

Role projections (RED-ROLE) are evaluated to equivalent query expressions. An equivalent query for a role projection is the query taking one argument (the term from which the role is selected) and selecting for the particular role. For queries themselves, the knowledge base (in practice, this would commonly be represented

x, y, z	(variable)
i	(IRI)
$v ::= \ldots$	(value)
$\mid$ **iri** i	(literal IRI)
$s, t, u ::= \ldots$	(term)
$\mid$ **sparql** q	(query)
$\mid$ **strictsparql** q	(strictly validated query)
$\mid t.R$	(role projection)
$\mid t$ **match** { $\overline{case}$ **case** _ => t }	(type case)
$case ::=$	(case expression)
case $x : C$ => t	(type case)
$S, T, U ::= \ldots$	(type)
$\mid C$	(concept type)
$R ::=$	(role expression)
i	(atomic role)
$\mid R^-$	(inverse role)
$C, D ::=$	(concept expression)
$\{i\}$	(nominal concept)
$\mid i$	(atomic concept)
$\mid \top$	(top)
$\mid \bot$	(bottom)
$\mid \neg C$	(negation)
$\mid C \sqcap D$	(intersection)
$\mid C \sqcup D$	(union)
$\mid \exists R.C$	(existential quantification)
$\mid \forall R.C$	(universal quantification)
$\alpha, \beta ::=$	(pattern element)
$?x$	(SPARQL variable)
$\mid t$	(term)
$pattern ::=$	(query patter)
$\alpha : C$	(concept assertion)
$\mid (i, \alpha) : R$	(from-lit role)
$\mid (\alpha, i) : R$	(to-lit role)
$\mid (\alpha, \beta) : R$	(role assertion)
$q, r ::=$	(query expression)
$pattern$	(query pattern)
$\mid q$. r	(conjunction)
$\mid q$ **UNION** r	(union)
$\mid q$ **MINUS** r	(minus)
$\mid q$ **OPTIONAL** r	(optional)

Figure 3 Syntax extensions defined by DOTSpa.

(RED-ROLE)
$t.R \to \textbf{strictsparql}\ (t, ?x) : R$

(RED-QUERY)
$\textbf{sparql}\ q \to \sigma([\![q]\!]^*_{\mathcal{K}})$

(RED-STRICT-QUERY)
$\textbf{strictsparql}\ q \to \sigma([\![q]\!]^*_{\mathcal{K}})$

(RED-DEFAULT)
$v\ \textbf{match}\ \{\textbf{case}\ \_\ \texttt{=>}\ t\} \to t$

(RED-MATCH)
$$\frac{t \to t'}{t\ \textbf{match}\ \{\overline{case}\} \to t'\ \textbf{match}\ \{\overline{case}\}}$$

(RED-CASE-S)
$$\frac{\mathcal{K} \models \{i\} \sqsubseteq C}{\begin{array}{l} i\ \textbf{match}\ \{ \\ \quad \textbf{case}\ x : C\ \texttt{=>}\ t \\ \quad \ldots \\ \quad \textbf{case}\ \_\ \texttt{=>}\ u \\ \} \end{array} \to [y \mapsto i]\, t}$$

(RED-CASE-F)
$$\frac{\mathcal{K} \not\models \{i\} \sqsubseteq C}{\begin{array}{l} i\ \textbf{match}\ \{ \\ \quad \textbf{case}\ x : C\ \texttt{=>}\ t \\ \quad \textbf{case}\ y : D\ \texttt{=>}\ s \\ \quad \ldots \\ \quad \textbf{case}\ \_\ \texttt{=>}\ u \\ \} \end{array} \to \begin{array}{l} i\ \textbf{match}\ \{ \\ \quad \textbf{case}\ y : D\ \texttt{=>}\ s \\ \quad \ldots \\ \quad \textbf{case}\ \_\ \texttt{=>}\ u \\ \} \end{array}}$$

Figure 4 Extended reduction rules.

by a SPARQL triple store) has to be consulted to obtain the solution sequence $[\![q]\!]_{\mathcal{K}}$. For brevity we omit reduction rules for terms embedded in queries. Such terms are assumed to be reduced via the normal reduction rules by $[\![q]\!]^*_{\mathcal{K}}$, which is otherwise based on the previously defined $[\![q]\!]_{\mathcal{K}}$ (Section 2). The query is then mapped to a implementation specific representation via σ. There is no difference in the evaluation of strict (RED-STRICT-QUERY) and non-strict queries.

After reducing the matched-on term of type cases (RED-MATCH), the different cases are tried in order: If the value is an IRI and has the respective concept expression type (RED-CASE-S), relying on judgments from the knowledge base, the match evaluates to the respective term, with substituted variable. If the matched value does not have the concept expression type (RED-CASE-F), the case is removed. For the single default case, the match expression evaluates to the default expressions term (RED-DEFAULT).

The type assignment and subtyping rules unique to DOTSpa are given in Figure 5. In order to assign the type of match expressions, the least upper bound (lub) of the types of all its branches is used (T-CASE). The lub of concept expression types is defined as the union of concepts ($\text{lub}(C, D) := C \sqcup D$). This definition extends recursively to any arity. Literal IRIs have a nominal concept type, based on the IRI itself (T-IRI). There exists a single subtyping rule for concept expression types: Two concept expression types are in the <: relation, if the respective concepts can be shown to be in a subsumptive relationship in context of the knowledge base (<:-CONCEPT).

In order to type queries (T-QUERY) and (T-STRICT-QUERY), the concept expressions for all its variables have to be inferred. Since queries may also contain arbitrary terms, but the algorithm for inference in Section 3 can only deal with SPARQL variables, we

$$\frac{}{\Gamma \vdash \textbf{iri}\ i : \{\ i\ \}}\ \text{(T-IRI)} \qquad \frac{\mathscr{K} \models C \sqsubseteq D}{\Gamma \vdash C <: D}\ (<:\text{-CONCEPT})$$

$$\frac{\Gamma \vdash t_n : T_n \qquad \Gamma, x_i : C_i \vdash t_i : T_i \text{ for } i = 1, \ldots, n-1}{\begin{array}{l}\Gamma \vdash i\ \textbf{match}\ \{ \\ \quad \textbf{case}\ x_1 : C_1\ \texttt{=>}\ t_1 \\ \quad \ldots \\ \quad \textbf{case}\ x_{n-1} : C_{n-1}\ \texttt{=>}\ t_{n-1} \\ \quad \textbf{case}\ \_\ \texttt{=>}\ t_n \\ \}\end{array} \rightarrow lub(T_1, \ldots, T_n)}\ \text{(T-CASE)}$$

$$\frac{q : \phi \qquad \forall x \in \text{vars}(q) : \mathscr{K} \models \phi(x) \not\equiv \bot \qquad \forall t \in \text{terms}(q) : \Gamma \vdash t : C \wedge \mathscr{K} \models C \sqsubseteq \phi(t)}{\Gamma \vdash \textbf{strictsparql}\ q : \sigma_T(\phi \text{ where } \forall t \in \text{terms}(q) : \phi(t) \text{ is replaced by } C)}\ \text{(T-STRICT-QUERY)}$$

$$\frac{q : \phi \qquad \forall x \in \text{vars}(q) : \mathscr{K} \models \phi(x) \not\equiv \bot \qquad \forall t \in \text{terms}(q) : \Gamma \vdash t : C \wedge \mathscr{K} \models \phi(t) \sqcap C \not\equiv \bot}{\Gamma \vdash \textbf{sparql}\ q : \sigma_T(\phi)}\ \text{(T-QUERY)}$$

$$\frac{((t, ?x) : R) : \phi \qquad \forall x \in \text{vars}(q) : \mathscr{K} \models \phi(x) \not\equiv \bot \qquad \Gamma \vdash t : C \wedge \mathscr{K} \models C \sqsubseteq \phi(t)}{\Gamma \vdash t.R : \sigma_T([\phi(?t) \mapsto C]\ \phi)}\ \text{(T-ROLE)}$$

Figure 5 Extended type assignment and subtyping rules.

map these terms to fresh SPARQL variables before typing the query. Then the mapping ϕ can be built according to the inference algorithm. In a slight abuse of notation, we use terms and the fresh variables they map to interchangeably. We also define vars(q) and terms(q) to refer to all variables and terms occuring in q, respectively. In order to validate a query, all concept expressions inferred for occurring variables x must be satisfiable (i.e., not equivalent to $\bot$). Otherwise, the query can be rejected as always empty. The second validation step varies for the strict and non-strict variants: For strict queries (T-STRICT-QUERY) the concept expression types C of the query-embedded terms t must be subsumed by the inferred types $\phi(t)$ for the matching, freshly introduced SPARQL variables. In the final type, the inferred types are then replaced by the (more specific) concept types C. For non-strict queries (T-QUERY) it suffices, that the intersection of C and $\phi(t)$ is satisfiable as well. The final result type is obtained by a function σ_T, taking the concept expression types as input. The precise type (much like the values constructed by σ) is not specified for DOTSpa and

depends on the implementation. The approach for role projections (T-ROLE) is the same as for strict queries (with one argument).

Example Consider the query **sparql** $(t, ?x)$: *takesCourse*, where t is a term with concept expression type *Professor*. For this non-strict query, it suffices that the knowledge base does not explicitly state that professors may never take courses (i.e., $\exists takesCourse.\top \sqcap Professor \not\equiv \bot$). Under this condition, the query is valid. For the respective strict query (**strictsparql** $(t, ?x)$: *takesCourse*), however, it must be possible to prove that all professors do in fact always take courses ($Professor \sqsubseteq takesCourse.\top$). Assuming that this is not true given the knowledge base, the query is not valid under strict SPARQL validation. If the argument was of type *Student* instead, the query would be valid. Then, the inferred type for the introduced variable for t could be substituted by the more precise type *Student*, in turn simplifing the type for $?x$. For common ontologies, this second approach can be too strict of a requirement.

5 Instantiating the DOTSpa Framework

DOTSpa is a general language extension framework for introducting querying and a type system for semantic data into programming. We provide a specific implementation called ScaSpa, which implements the DOTSpa approach in the functional programming language Scala. The integration of concept expressions and SPARQL into practical programming technologies, such as the Scala language, introduces several issues. From a practical point of view, the T-Box and A-Box of a knowledge base are often separated. For the T-Box, we rely on ontology reasoners, which are optimized for fast T-Box reasoning. Data however is best stored in a triple store. Both, ontology reasoner and triple store, are part of ScaSpa in terms of the underlying language integration and architecture.

Merging of Three Languages DL concept expressions as well as the SPARQL query language must be syntactically integrated into the host language Scala. We therefore face similar issues as identified by [22, 44, 62], in particular with respect to scoping and the interaction between the languages, such as unquoting of Scala variables in SPARQL queries.

Knowledge Base Integration into Static Type Checking DL concept expressions create a new form of types that come with their own set of rules in terms of subtyping, creating an amalgamated type system. The behavior of these new form of types is defined through an ontology reasoner, which must be integrated into the type checking process of Scala, so it can provide judgments to the type checker. This is comparable to the integration of Coq into ML as described by [25].

Runtime Checks Objects in a knowledge base do not have a principal type [58] except for the concept that consists only of the object itself. Additionally, knowledge is assumed to be incomplete. Our approach is similar to type-based filters, for example

Listing 1 A function querying for all research groups that are sub-organizations of a given organization.

```
def researchGroups(other: `:Organization`): List[`:ResearchGroup`] =
  sparql"""
    SELECT ?rq WHERE {
      ?rq a :ResearchGroup .
      ?rq :subOrganizationOf $other.
    }
  """

def supervises(chair: `:Chair`): List[`:ResearchGroup`] = {
  val deps = chair.`:headOf`
  if (deps.nonEmpty) researchGroups(deps.head)
  else Nil
}
```

Table 1 Type checks occurring in the program in Listing 1 when applying `supervises` with `c`, where `c` has the (concept expression) type `` `:Chair` ``.

Line	Case	Type check	Kind
	call supervises	$\text{Chair} \sqsubseteq \text{Chair}$	(E-SUB)
10	role projection	$\text{Chair} \sqsubseteq \exists \text{headOf}.\top$	(E-ACC)
11	application	$\exists \text{headOf}^{-}.\text{Chair} \sqsubseteq \text{Organization}$	(E-SUB)
2	validation (`rq`)	$\exists \text{subOrganizationOf}.\top \sqcap \text{ResearchGroup} \not\equiv \bot$	(E-SAT)
2	validation (`other`)	$\exists \text{subOrganizationOf}^{-}.\text{ResearchGroup} \sqcap \text{Organization} \not\equiv \bot$	(E-SAT)
1	return type	$\exists \text{subOrganizationOf}.\top \sqcap \text{ResearchGroup} \sqsubseteq \text{ResearchGroup}$	(E-SUB)
9	return type	$\text{ResearchGroup} \sqsubseteq \text{ResearchGroup}$	(E-SUB)

as in LINQ [50] or a type dispatch [27]. However, in our case, such filters or type dispatches require a translation into an equivalent query answered by the triple store.

Application Scenario Our approach enables ontology-based type checking. In addition to standard errors prevented by type checking, such as typos, this allows for the detection of three kinds of common errors:

E-SAT Usatisfiable queries, meaning that there is no possible database instance that can answer the query.

E-ACC Access on properties that are not known to exist for a value.

E-SUB Unintended values by the programmer, as expressed by type annotations.

Listing 2 Syntax and internal representation of concept expression types.

```
1 def empl: `:Person ⊓ ∃:worksFor.:Organization` // syntax
2 def empl: DLType @dl(":Person ⊓ ∃:worksFor.:Organization") // internal
```

As an example of this, consider a management application for a university. Such a program may include a function *researchGroups* : *Organization* → `List[`*ResearchGroup*`]` that, given an *Organization*, lists all *ResearchGroups* that are direct sub-organizations. This function can be implemented using a simple SPARQL query, splicing in the given organization. Type checking ensures that the query is satisfiable (E-SAT). Lines 2–7 of Listing 1 show an example of such a query. One important feature is that the types *Organization* and *ResearchGroup* directly represent the concept expressions as defined by the axioms of Figure 1. It guarantees that the function can only be applied to values which are some form of *Organization*, as intended by the developer (E-SUB). Another example of this is the *supervises* : *Chair* → `List[`*ResearchGroup*`]` function (lines 9–13). It takes values of the concept *Chair* as input.

In this function the developer wants to access the role *headOf*. The type checking process has to show that all *Chairs* have such a role according to the ontology, preventing the access to non-existing properties (E-ACC). For simplicity, we take the head of the returned list and use the previously defined function *researchGroups* to find all *ResearchGroups*. This is possible as it is known that a *Chair* is the head of a *Department*, which is a special kind of *Organization* (E-SUB). All type checks and validation steps occurring in this example in Listing 1 are summarized in Table 1.

6 Architecture and Implementation of ScaSpa

ScaSpa is a strict extension of Scala. In particular, the type checking process is extended, so that an ontology reasoner can be used for dealing with concept expressions. As this process necessarily relies on typing information, preprocessing in the form of simple desugaring of extended constructs into standard Scala is not sufficient. Instead, we rely on the extension interface of the Scala compiler. Figure 6 gives an overview of the integration. While we focus on Scala, its primary components can also be understood as a general architecture for transferring DOTSpa into practice.

Parser We use a staged parsing approach. Initially, DL concept expressions and SPARQL queries are essentially parsed as strings (through the Scala backquote and StringContext features). Syntactic validity of concept expressions and SPARQL queries is ignored at this stage—instead, the Scala parser creates a standard AST. Later stages recover these constructs through AST traversals. In the DL parser stage, syntactic validity and satisfiability of concept expressions is checked, before erasing them to a base type. The concrete concept expression lives on through a static annotation on this type (see Listing 2). Such static annotations also persist in the (metadata of) the generated byte code, preserving incremental and separate compilability. The concept

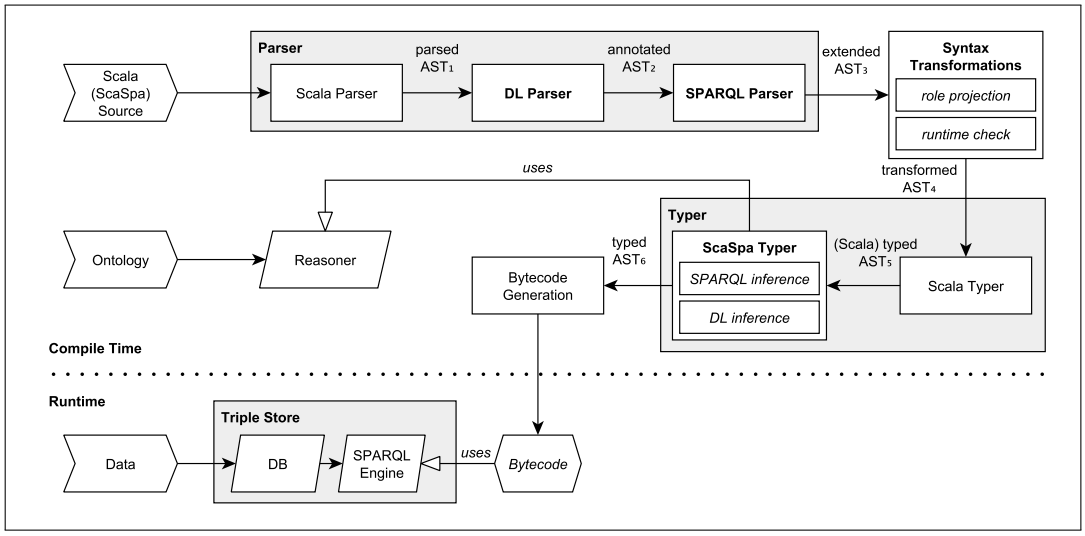

Figure 6 Architectural model of ScaSpa.
Nodes are compilation stages (rectangle), summarized stages (shaded rectangle), artifacts (arrow) and external components (parallelogram). Arrows are dataflow (filled heads) and dependency (unfilled heads).

Listing 3 SPARQL query syntax relies on the transformation of `StringContext`.

```
def profs = sparql"SELECT ?x WHERE { $empl :worksFor ?x }" // syntax
def profs = // internal
  StringContext("SELECT ?x WHERE {", ":worksFor ?x }").sparql(empl)
```

expressions themselves use standard DL syntax, with the addition that concepts are represented by IRIs. As in SPARQL, a prefix alias can be defined and used. Our examples use the default prefix `':'` for the Lehigh University Benchmark ontology.

Parsing of SPARQL queries, which might contain unquoted Scala expressions, works similarly—the queries are checked for syntactic validity and type annotations are added (see Listing 3). Internally, such a query is represented by the `StringContext` class, which in turn exploits Scala's built-in syntax transformation for prefixed strings. The same feature also handles the insertion of context arguments using `'$'`. For queries, the parser attaches the general `DLType`, while the specific concept expressions are inferred later, as they might depend on the types of query arguments.

Syntax Transformations As a next step, role projections (Listing 1, line 10) and type cases are simplified into queries. For role projections, this reduction was already defined in Section 4. Due to the separation of the T-Box and A-Box in reasoner (compile time) and triple store (runtime), runtime subtyping has to be resolved using the triple store. To this end, runtime checks are transformed into an actual `instance-of` test based on the base type and a SPARQL `ASK` query—a special form of SPARQL query that evaluates to either true or false. The only limitations of this approach is an over-approximation of results due to the differing notions of negation

Listing 4 Least upper bound inference for concept expression types is the union of concepts.

```
val prof: `:Professor` = // ...
val resa: `:ResearchAssistant` = // ...

val lst: List[`:Professor ⊔ :ResearchAssistant`] = List(prof, resa)
```

Listing 5 Concrete ScaSpa implementation of the σ and σ_T functions.

```
def employment: List[(`:Person`, `:Organization`)] =
  sparql" SELECT ?p ?c WHERE { ?p :worksFor ?c } "
```

existing between description logics and SPARQL. This was previously observed for the type inference of *MINUS* queries.

Typer After parsing and performing the syntax transformations, the AST contains only valid Scala, including the base type `DLType` with static annotations for concept expression types. This allows the standard Scala typer to do local type inferencing as well as type checking based on `DLType`. Since there are no more extended constructs, the typed AST is produced in a normal manner. Additional type checker rules for concept expressions and SPARQL queries are implemented in a phase after the Scala typer, relying on the propagation of the base type. The ontology reasoner (ScaSpa uses HermiT [51]) and the actual ontology containing the data descriptions are used during this phase. As OWL includes a namespace feature to distinguish concept expressions, namespace managing is also taken care of by the ontology reasoner.

In order to perform type checking and inference according to the rules defined in Section 4, the typed AST is traversed again. During this traversal, the ScaSpa typer performs type checks on, and propagates the, static annotations where the base type was inferred by the Scala typer. A notable difference between the DOTSpa formalization and ScaSpa is that the latter uses a T-Box only mode by default. In T-Box only mode, nominal types are instead estimated using $\top$ (e.g., in literal IRIs) if no explicit annotation is provided. This preserves the separation of T-Box and A-Box. In addition to the specified typing rules, some additional constructs of Scala have to be considered. This includes most importantly type parameters and related features. In order to infer concrete types for type parameters, the least upper bound is sometimes required. This is, for example, the case when inferring the type of if-expressions or constructors for which the same type parameter occurs multiple times (Listing 4). Additionally, Scala allows for the explicit definition of variances (invariant, covariant, contravariant) for type parameters and upper as well as lower bounds. All these features can be directly mapped to the (<:-CONCEPT) rule as defined in Section 4. Finally, DOTSpa requires implementations to provide a representation for queries (namely the σ and σ_T functions). In ScaSpa we use simple lists of tuples as shown in Listing 5.

The resulting AST represents a normal Scala program. Transformation into byte code is therefore a standard procedure. To evaluate queries at runtime, arguments are

converted to strings and spliced into the queries. In addition to arguments of concept expression types, ScaSpa supports a limited set of Scala types, which are mapped to appropriate XSD data types (e.g., `String` is mapped to `xsd:string`). Therefore it is necessary to take special care to escape the arguments, so that query injections can be avoided. The assembled queries are then handed to a triple store (where we employ Stardog [71]) for evaluation.

Limitations Ad hoc polymorphism in the form of method overloading and implicit parameters (used by Scala's notion of type classes), are implemented by compiler internals not exposed through the compiler extension interface. Since all concept expression types are internally represented by the same base type, neither the resolution of overloaded methods nor the implicit search can handle them. Workarounds, such as custom dispatch at runtime or patching the compiler directly, might be possible solutions for this limitation, but remain as future work.

7 Evaluation

ScaSpa aims at both increasing type safety and reducing complexity by providing an advanced integration of semantic data in Scala. In particular, it aims to reduce the overhead that results from the boilerplate code commonly required to handle querying, including query construction and execution, while not increasing complexity due to any type system related features. Type safety of the underlying approach was already shown with λ_{DL}. In order to show the feasibility of ScaSpa, we compare implementations of two small use cases between our approach and a traditional RDF library. To this end, we chose the `banana-rdf` [5] framework. This library for working with RDF and SPARQL is implemented in Scala and therefore allows for an equal basis of comparison, eluding any programming language related differences while relying on the prevalent Jena [13] framework internally. We compare implementations of both approaches using the Halstead complexity measure [32]. While Halstead is sometimes criticized for its simplicity [19], it allows us to measure the impact of ScaSpa on the difficulty and effort to understand and write these programs. Similar complexity metrics are not suitable for this purpose: Cyclomatic complexity [48] does not apply in our use case, since control flow is not primarily impacted by the change in querying framework. Similar arguments can be made for the information flow [36] or function point [2] metrics, while common metrics for object oriented programs [14] are not applicable to the functional style or the extend of our use cases.

Definition 1 *Let n_1 be the number of distinct operators, N_1 the total number of operators, n_2 the number of distinct operands and N_2 the total number of operands. Then Halstead difficulty and effort are defined as follows:*

Program vocabulary	$n = n_1 + n_2$	**Program length**	$N = N_1 + N_2$
Volume	$V = N \times log_2(n)$		
Difficulty	$D = \frac{n_1}{2} \times \frac{N_2}{n_2}$	**Effort**	$E = D \times V$

In order to calculate the Halstead metric, we use the following definitions for operators and operands: We count reserved keywords (such as `new` or `if`, including the related parentheses) and operations like member access, assignment or function application as operators. We count all other identifiers, type names or constants as operands. For SPARQL queries, we count `OPTION` and `SELECT` as one operator, `PREFIX` as one operator with two operands and a single triple pattern as one operator with three operands.

We measure difficulty and effort according to the Halstead metric. We expect ScaSpa to outperform `banana-rdf` in both of these aspects. In addition, we measure the size of the resulting compilation units, time necessary for compilation as well as runtime performance. We expect ScaSpa to perform similarly to `banana-rdf`.

7.1 Use Cases

We define two use cases, covering the various aspects of ScaSpa. As a data source, we use the auto-generated data provided by the Lehigh University Benchmark [31]. While the benchmark is designed for testing the performance of OWL knowledge systems, it also provides an interesting ontology with familiar hierarchies for our setting.

Our use cases are inspired by queries used in the benchmark, particularly queries that involve reasoning. The queries were adapted (e.g., by substituting IRI literals with variables from the program context) to cover all features of ScaSpa. The use cases also provide opportunities for the usage of literal IRIs, role projections and type cases, covering the core elements of ScaSpa. Additionally, the use cases include some of the most frequently used types of SPARQL queries as identified in [66]. In particular, queries that can be simplified to role projections in ScaSpa are the most common in practice.

Use Case 1: A function for returning research groups of organizations

Our first use case combines two common tasks: The definition of functions returning the results of queries and the inclusion of arguments from the programming context into such queries.

R1 The program shall define a function, which takes an organization as its argument and returns all research groups that are sub-organizations of it.

R2 The program shall define a function, which takes a department chair as its argument and returns all research groups of the department this chair supervises.

`banana-rdf` Implementation The `banana-rdf` implementation of use case 1 (Listing 6) defines the method `researchGroups`, taking as its argument the target organization (as a string) and returning a generic solution sequence. This method constructs the final query by string interpolation and parses it to obtain a query object. The SPARQL query itself is a raw string and consists of two triple patterns, restricting a variable to be both a research group and sub-organization of the argument. Finally, the parsed query can be executed against the triple store and the result is returned.

Listing 6 Use case 1 (`banana-rdf` implementation).

```
def researchGroups(other: String): Try[Rdf#Solutions] = for {
   q <- parseSelect(
        s"""
          PREFIX : <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
          SELECT ?org WHERE {
            ?org a :ResearchGroup .
            ?org :subOrganizationOf <$other>
          }""");
   r <- sparql.executeSelect(q)
  } yield r

def supervises(chair: String): Try[Rdf#Solutions] = for {
   q <- parseSelect(
        s"""
          PREFIX : <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
          SELECT ?org WHERE {
            <$chair> :headOf ?org
          }""");
   deps <- sparql.executeSelect(q);
   depit <- Try(deps.iterator.next());
   dep <- depit("?org");
   uridep <- dep.as[Rdf#URI];
   r <- researchGroups(uridep.toString)
  } yield r
```

Table 2 Metrics for use case 1 (full results in Appendix A).

	banana-rdf	**ScaSpa**
Difficulty D	12.10	8.25
Effort E	6983.42	1686.80
Compilation unit size s	27.7 KiB	9.8 KiB
Compilation time t_c	1.371 s	1.914 s (0.947 s)
Execution time t_r	0.060 s	0.063 s

The method `supervises` is defined in a similar way. The query result is used to call `researchGroups` with the appropriate department. Since various steps, including parsing the query or the selection of ?org can fail, everything is wrapped in `Try`.

ScaSpa Implementation The ScaSpa solution (Listing 1) was used as an example in Section 5. It uses the same queries as the `banana-rdf` implementation, even though one is expressed as a role projection.

Comparison For the first use case, we obtain the results as shown in Table 4, displaying an improvement from `banana-rdf` to ScaSpa for both difficulty and effort. The

compilation unit is smaller in the case of ScaSpa. This is in part due to some constant overhead in `banana-rdf`. The difference in compilation time is expected as we rely on several additional AST traversals during compilation. The fraction of compilation time our newly introduced phases take is given in parentheses. Additionally, we have some constant overhead such as instantiation of the ontology reasoner. In terms of execution time, ScaSpa is marginally slower. This might be due to the fact that, before executing the query, we process any values being spliced into our query—a step that `banana-rdf` does not do.

Use Case 2: Finding all professors including department chairs
The second use case poses the task of processing and displaying query results, while performing additional actions (output in this case) for certain results.

R1 The program shall return all professors working for a given department.

R2 The program shall process the result and display all professors, marking the department head as "chair".

`banana-rdf` Implementation The `banana-rdf` implementation for the second use case (Listing 7 in the appendix) requires the same steps as for use case 1, including the manual parsing and constructing of the query. In this case, the query includes an optional expression, returning two variables: The professor and the department this professor is head of, if any. This is not the only possible definition of a query—alternatives include the possibility to bind variables to certain values. However, it is the least complex query for the task. After executing the query, the results are manually accessed by selecting the respective variable and casting it to `URI`. If the department is defined, the professor is marked as chair.

ScaSpa Implementation As we try to identify the most idiomatic style for each approach, our ScaSpa implementation (Listing 8 in the appendix) differs from the implementation using `banana-rdf`. While the same query could have been used in both solutions, our ScaSpa implementation leverages the available expressiveness of type case expressions to find the department chair, after querying for all professors. This also allows us to generalize the `professorsAndChairs` function to simply return all `professors` of a department.

Comparison According to the Halstead metric (see Table 5), the ScaSpa solution can again shown to be less difficult and require less effort to implement than the solution based on `banana-rdf`. The compilation unit size is again larger for `banana-rdf`, while compilation time increases for ScaSpa. For this use case, we observe a larger increase in execution time for ScaSpa, due to the differing approach: The match-expression gets desugared into a (simple) SPARQL ASK query, which is executed for each result of the original query. These additional query executions are responsible for the larger overhead.

Table 3 Metrics for use case 2 (full results in Appendix B).

	`banana-rdf`	ScaSpa
Difficulty D	16.55	7.50
Effort E	9806.66	1375.49
Compilation unit size s	25.7 KiB	9.2 KiB
Compilation time t_c	1.428 s	1.919 s (0.989 s)
Execution time t_r	0.265 s	0.317 s

7.2 Discussion of Evaluation Results

Our initial assumption, that ScaSpa outperforms `banana-rdf` in both Halstead difficulty and effort is supported by the results of the metric: Both use cases demonstrate that ScaSpa is easier to understand and takes less effort to implement. We observe no increase in compilation unit size. In terms of execution time, ScaSpa performs slightly below `banana-rdf` for the already discussed reasons, in cases where the general approach is the same. Match expressions, while resulting in a larger runtime overhead, are optional and as such provide a trade off between reduced complexity and runtime performance. While compilation time increases more significantly for ScaSpa, only part of this increase scales linearly with the program size: Almost half of the time spent in the newly introduced phases (0.45 s on average) is required to instantiate the reasoner and therefore constant.

8 Related Work

DOTSpa and the ScaSpa implementation are generally related to three larger areas: Language extensions, integration of semantic data as well as empirical language evaluation.

Language Extension Extending programming languages is a long standing topic [22, 44, 62]. Numerous systems, such as TemplateHaskell [64], Racket [24], SugarJ [22], LINQ [50] and Scala macros [12] provide syntactic extensions based on AST transformations. With DOTSpa, we require an extended type checker, so approaches relying solely on transformations of the AST are not suitable. Instead, the conceptual framework proposed in [47] is closer to our approach. Other systems for compiler extensions include Polyglot [52] and ExtendJ [20]. In order to stay within the standard Scala pipeline, rather than creating a new compiler, we chose Scala compiler extensions [65].

ScaSpa relies on an amalgamation of two type systems—one for the normal programming language constructs and one for DL concept expressions. The idea of pluggable type systems [11, 56] that allow for new type systems being layered on top of existing ones has some similarities. Indeed, ScaSpa can be seen as an additional layer on

top of the Scala type system. However, the approach is different from ours, since we integrate an ontology reasoner providing the type system judgements. Similarly, open type systems such as provided by the JVM language Gosu [49] allow for the definition of new base types, but do not involve the problem of reasoner integration. Type providers [15, 46, 68] follow a goal similar to ScaSpa—bridging the gap between programming language and information sources. However, in essence they aim at mappings. A mapping, however, either completely duplicates (if possible) or approximates reasoning behavior, which is undesirable.

Another related direction in bridging programming language and information source are type systems that are extended for particular kinds of data. Examples include [9] for relational data, [53, 74] for object oriented databases and [8] for XML data. Albeit not being language extensions, the regular expression types provided by CDuce [7] and XDuce [40] are related to ScaSpa due to their unique form of types. Refinement type systems, e.g., provided by F* [67], are somewhat closer to ScaSpa, although typically focused on pre- and postconditions of functions. In contrast, DL concept expressions are logical formulae over nominal and structural type properties. Their defined types are subject to DL reasoning during type checking. As such, ScaSpa is much closer to the integration of Coq in OCaml as described by [25]. In particular to the idea of using the theorem prover—or in our case the ontology reasoner—in the type checking process.

RDF and Ontology Integration The problem of accessing and integrating RDF data in programming languages has been recognized as a challenge in various works. Examples for untyped frameworks include banana-rdf [5], the OWL API [38], Jena [13] and RDF4J [60]. Such frameworks generally provide abstractions on the meta-level, for example in Jena with Java classes such `OntClass` to represent OWL or RDFS classes. While this reflection-like approach might be suitable for developing ontology based tools, it is lacking when working with concrete ontologies [29]. In particular, any correctness of the program related to the data is left completely in the hand of the programmer.

Approaches that create mappings between ontologies and, for example, the object model of object oriented languages, can offer at least some form of verification. Existing mapping frameworks include ActiveRDF [54], Alibaba [70], Owl2Java [41], Jastor [69], RDFReactor [61], OntologyBeanGenerator [1], Àgogo [57] and LITEQ [46]. However, mapping approaches are problematic due to the mixture of nominal and structural typing, as well as implicit relations such as the relation between *ResearchAssistant* and *Employee* in Figure 1.

Some implementations with a deeper integration into programming languages are available. Zhi# [55] extends the type system of C# for OWL and XSD types. The main technical difference is that ScaSpa uses an ontology reasoner in the type checker, allowing for the handling of inferred data. SWOBE [30] provides a typed integration of SPARQL into Java through a precompilation phase—but is limited to primitive datatypes, IRIs and a triple-based datatype. Additionally, some custom languages exist, that use static type-checking for querying and light scripting to avoid runtime

errors [16, 17]. However, the types are again limited in these cases, as they only consider explicitly given statements.

Empirical Evaluation of Programming Languages Programming languages can be evaluated in several ways: Language definitions can be evaluated for their theoretical properties, e.g., with regard to type safety [4, 58]. Languages can also be evaluated empirically, e.g., through controlled experiments. This way the effects of whole concepts, e.g., Aspect Oriented Programming [33], or smaller parts of a language, e.g. the effects of generic types [37] or static typing [21, 34], can be evaluated. We plan on conducting similar studies in the future to further to show the reduced complexity according to our metrics-based evaluation has an effect in practice. Focussing on the program itself as well as derived artifacts is another common approach in empirical software evaluation. Large corpora of programs can e.g., be analyzed for common structures [6] or micro patterns [26]. Likewise, derived artifacts, such as Java Bytecode may also be studied [18].

Direct comparisons of various approaches however often rely on custom domain-dependent tasks which are then evaluated with regard to certain aspects. This has, for example, been done for generic programming in Haskell [63] with aspects such as ease of learning as well as the overhead of using the library. Other examples include the comparison of language workbenches [23] in terms of their features or the comparison of lazy functional languages [35] in terms of performance. The underlying idea of comparing implementations based on domain-dependent tasks with respect to certain criteria can be found quite often in literature as e.g., highlighted by [46] or [43]. The evaluation of ScaSpa is directly inspired by these approaches. We focus our evaluation on domain-dependent tasks and compare the libraries with respect to effort in writing and ease of understanding the programs, as well as practical aspects such as size and performance.

9 Summary and Future Work

In this paper we presented DOTSpa—a deep integration of semantic data into practical programming. This is achieved by providing DL concept expressions as a new form of types and via the deep, typed integration of the SPARQL query language. Further, we implement this approach as the ScaSpa extension for Scala. This implementation is based on a staged parsing approach, type judgements provided by an ontology reasoner to the type system, as well as type erasure. We also demonstrated how our approach reduces complexity through a metrics-based evaluation.

Our work can be extended in several directions. Strictly distinguishing between the ontology reasoner at compile time and the triple store at runtime ensures a good runtime performance of match-expressions on concept expressions. As mentioned before, it introduces an overestimation when used in combination with negation. We plan to investigate into performant ways of combining the ontology reasoner and triple store for cases in which negation is involved. Another technical limitation we already mentioned is ad hoc polymorphism. As we erase type information, standard

ad hoc polymorphism such as method overloading mechanisms provided by Scala do not work. We plan to investigate possible solutions to this.

As of now, ScaSpa does not provide any support for tooling beyond compilation. Code completion and IDE support is of high interest to us. In particular, code completion on DL concept expressions and SPARQL queries would be useful. Support for tooling opens up another direction of future work—an evaluation using user studies. Even though our metrics-based evaluation show a reduction in complexity, user studies can evaluate the impact of features provided by ScaSpa on real users.

References

[1] Chris van Aart. *OntologyBeanGenerator*. https://protegewiki.stanford.edu/wiki/OntologyBeanGenerator. Sept. 27, 2018. (Visited on 2018-09-27).

[2] Allan J. Albrecht and John E. Gaffney Jr. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation". In: *IEEE Trans. Software Eng.* 9.6 (1983), pages 639–648. DOI: 10.1109/TSE.1983.235271.

[3] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. "The Essence of Dependent Object Types". In: *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. 2016, pages 249–272. DOI: 10.1007/978-3-319-30936-1_14.

[4] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. "Mechanized Metatheory for the Masses: The PoplMark Challenge". In: *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*. 2005, pages 50–65. DOI: 10.1007/11541868_4.

[5] banana-rdf. *banana-rdf*. https://github.com/banana-rdf/banana-rdf. (Visited on 2018-09-27).

[6] Gareth Baxter, Marcus R. Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan D. Tempero. "Understanding the shape of Java software". In: *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*. 2006, pages 397–412. DOI: 10.1145/1167473.1167507.

[7] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. "CDuce: an XML-centric general-purpose language". In: *SIGPLAN Notices* 38.9 (2003), pages 51–63. DOI: 10.1145/944746.944711.

[8] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. "The Essence of Data Access in Comega". In: *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*. 2005, pages 287–311. DOI: 10.1007/11531142_13.

[9] Gavin M. Bierman and Alisdair Stuart Wren. “First-Class Relationships in an Object-Oriented Language”. In: *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*. 2005, pages 262–286. DOI: 10.1007/11531142_12.

[10] Olivier Bodenreider, Barry Smith, Anand Kumar, and Anita Burgun. “Investigating subsumption in SNOMED CT: An exploration into large description logic-based biomedical terminologies”. In: *Artificial Intelligence in Medicine* 39.3 (2007), pages 183–195. DOI: 10.1016/j.artmed.2006.12.003.

[11] Gilad Bracha. “Pluggable Type Systems”. In: *OOPSLA Workshop on Revival of Dynamic Languages*. 2004.

[12] Eugene Burmako. “Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming”. In: *Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013*. 2013, 3:1–3:10. DOI: 10.1145/2489837.2489840.

[13] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. “Jena: implementing the semantic web recommendations”. In: *Proceedings of the 13th international conference on World Wide Web - Alternate Track Papers & Posters, WWW 2004, New York, NY, USA, May 17-20, 2004*. 2004, pages 74–83. DOI: 10.1145/1013367.1013381.

[14] Shyam R. Chidamber and Chris F. Kemerer. “Towards a Metrics Suite for Object Oriented Design”. In: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’91), Sixth Annual Conference, Phoenix, Arizona, USA, October 6-11, 1991, Proceedings.* 1991, pages 197–211. DOI: 10.1145/117954.117970.

[15] David Raymond Christiansen. “Dependent type providers”. In: *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming, WGP 2013, Boston, Massachusetts, USA, September 28, 2013*. 2013, pages 25–34. DOI: 10.1145/2502488.2502495.

[16] Gabriel Ciobanu, Ross Horne, and Vladimiro Sassone. “Descriptive Types for Linked Data Resources”. In: *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*. 2014, pages 1–25. DOI: 10.1007/978-3-662-46823-4_1.

[17] Gabriel Ciobanu, Ross Horne, and Vladimiro Sassone. “Minimal type inference for Linked Data consumers”. In: *J. Log. Algebr. Meth. Program.* 84.4 (2015), pages 485–504. DOI: 10.1016/j.jlamp.2014.12.005.

[18] Christian S. Collberg, Ginger Myles, and Michael Stepp. “An empirical study of Java bytecode programs”. In: *Softw., Pract. Exper.* 37.6 (2007), pages 581–641. DOI: 10.1002/spe.776.

[19] Bill Curtis, Sylvia B. Sheppard, Phil Milliman, M. A. Borst, and Tom Love. “Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics”. In: *IEEE Trans. Software Eng.* 5.2 (1979), pages 96–104. DOI: 10.1109/TSE.1979.234165.

[20] Torbjörn Ekman and Görel Hedin. "The jastadd extensible java compiler". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. 2007, pages 1–18. DOI: 10.1145/1297027.1297029.

[21] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. "How do API documentation and static typing affect API usability?" In: *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 2014, pages 632–642. DOI: 10.1145/2568225.2568299.

[22] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. "SugarJ: library-based syntactic language extensibility". In: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. 2011, pages 391–406. DOI: 10.1145/2048066.2048099.

[23] Sebastian Erdweg et al. "Evaluating and comparing language workbenches: Existing results and benchmarks for the future". In: *Computer Languages, Systems & Structures* 44 (2015), pages 24–47. DOI: 10.1016/j.cl.2015.08.007.

[24] Matthew Flatt. "Creating languages in Racket". In: *Commun. ACM* 55.1 (2012), pages 48–56. DOI: 10.1145/2063176.2063195.

[25] Seth Fogarty, Emir Pasalic, Jeremy G. Siek, and Walid Taha. "Concoqtion: indexed types now!" In: *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*. 2007, pages 112–121. DOI: 10.1145/1244381.1244400.

[26] Joseph Gil and Itay Maman. "Micro patterns in Java code". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. 2005, pages 97–116. DOI: 10.1145/1094811.1094819.

[27] Neal Glew. "Type Dispatch for Named Hierarchical Types". In: *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999*. 1999, pages 172–182. DOI: 10.1145/317636.317797.

[28] Birte Glimm, Chimezie Ogbuji, Sandro Hawke, Ivan Herman, Bijan Persia, Axel Polleres, and Andy Seaborne. *SPARQL 1.1 Entailment Regimes*. W3C Rec. https://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/#OWLRDFBSEntRegime. July 7, 2018. (Visited on 2018-07-07).

[29] Neil M. Goldman. "Ontology-Oriented Programming: Static Typing for the Inconsistent Programmer". In: *The Semantic Web - ISWC 2003, Second International Semantic Web Conference, Sanibel Island, FL, USA, October 20-23, 2003, Proceedings*. 2003, pages 850–865. DOI: 10.1007/978-3-540-39718-2_54.

[30] Sven Groppe, Jana Neumann, and Volker Linnemann. “SWOBE - embedding the semantic web languages RDF, SPARQL and SPARUL into java for guaranteeing type safety, for checking the satisfiability of queries and for the determination of query result types”. In: *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*. 2009, pages 1239–1246. DOI: 10.1145/1529282.1529561.

[31] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. “LUBM: A benchmark for OWL knowledge base systems”. In: *J. Web Sem.* 3.2-3 (2005), pages 158–182. DOI: 10.1016/j.websem.2005.06.005.

[32] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977. ISBN: 0444002057.

[33] Stefan Hanenberg, Sebastian Kleinschmager, and Manuel Josupeit-Walter. “Does aspect-oriented programming increase the development speed for crosscutting code? An empirical study”. In: *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, October 15-16, 2009, Lake Buena Vista, Florida, USA*. 2009, pages 156–167. DOI: 10.1145/1671248.1671264.

[34] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. “An empirical study on the impact of static typing on software maintainability”. In: *Empirical Software Engineering* 19.5 (2014), pages 1335–1382. DOI: 10.1007/s10664-013-9289-1.

[35] Pieter H. Hartel and Koen Langendoen. “Benchmarking Implementations of Lazy Functional Languages”. In: *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*. 1993, pages 341–349. DOI: 10.1145/165180.165230.

[36] Sallie M. Henry and Dennis G. Kafura. “Software Structure Metrics Based on Information Flow”. In: *IEEE Trans. Software Eng.* 7.5 (1981), pages 510–518. DOI: 10.1109/TSE.1981.231113.

[37] Michael Hoppe and Stefan Hanenberg. “Do developers benefit from generic types?: an empirical comparison of generic and raw types in java”. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 2013, pages 457–474. DOI: 10.1145/2509136.2509528.

[38] Matthew Horridge and Sean Bechhofer. “The OWL API: A Java API for OWL ontologies”. In: *Semantic Web* 2.1 (2011), pages 11–21. DOI: 10.3233/SW-2011-0025.

[39] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. “From SHIQ and RDF to OWL: the making of a Web Ontology Language”. In: *J. Web Sem.* 1.1 (2003), pages 7–26. DOI: 10.1016/j.websem.2003.07.001. URL: https://doi.org/10.1016/j.websem.2003.07.001.

[40] Haruo Hosoya and Benjamin C. Pierce. "XDuce: A statically typed XML processing language". In: *ACM Trans. Internet Techn.* 3.2 (2003), pages 117–148. DOI: 10.1145/767193.767195.

[41] Aditya Kalyanpur, Daniel Jiménez Pastor, Steve Battle, and Julian A. Padget. "Automatic Mapping of OWL Ontologies into Java". In: *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004), Banff, Alberta, Canada, June 20-24, 2004*. 2004, pages 98–103.

[42] Ilianna Kollia, Birte Glimm, and Ian Horrocks. "SPARQL Query Answering over OWL Ontologies". In: *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29-June 2, 2011, Proceedings, Part I*. 2011, pages 382–396. DOI: 10.1007/978-3-642-21034-1_26.

[43] Jochen M. Küster, Shane Sendall, and Michael Wahler. "Comparing Two Model Transformation Approaches". In: *Proc. Workshop on OCL and Model Driven Engineering*. 2004.

[44] Ralf Lämmel. *Software languages: Syntax, semantics, and metaprogramming*. Springer, 2018. ISBN: 978-3-319-90798-7. DOI: 10.1007/978-3-319-90800-7.

[45] Martin Leinberger, Ralf Lämmel, and Steffen Staab. "The Essence of Functional Programming on Semantic Data". In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 2017, pages 750–776. DOI: 10.1007/978-3-662-54434-1_28.

[46] Martin Leinberger, Stefan Scheglmann, Ralf Lämmel, Steffen Staab, Matthias Thimm, and Evelyne Viegas. "Semantic Web Application Development with LITEQ". In: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II*. 2014, pages 212–227. DOI: 10.1007/978-3-319-11915-1_14.

[47] Florian Lorenzen and Sebastian Erdweg. "Sound type-dependent syntactic language extension". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, pages 204–216. DOI: 10.1145/2837614.2837644.

[48] Thomas J. McCabe. "A Complexity Measure". In: *IEEE Trans. Software Eng.* 2.4 (1976), pages 308–320. DOI: 10.1109/TSE.1976.233837.

[49] Scott McKinney. *Gosu's secret sauce: The open type system*. http://guidewiredevelopment.wordpress.com/2010/11/18/gosus-secret-sauce-the-open-type-system. Nov. 2010. (Visited on 2018-04-07).

[50] Erik Meijer, Brian Beckman, and Gavin M. Bierman. "LINQ: reconciling object, relations and XML in the .NET framework". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. 2006, page 706. DOI: 10.1145/1142473.1142552.

[51] Boris Motik, Bernardo Cuenca Grau, and Ulrike Sattler. "Structured objects in owl: representation and reasoning". In: *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*. 2008, pages 555–564. DOI: 10.1145/1367497.1367573.

[52] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. "Polyglot: An Extensible Compiler Framework for Java". In: *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. 2003, pages 138–152. DOI: 10.1007/3-540-36579-6_11.

[53] Atsushi Ohori, Peter Buneman, and Val Tannen. "Database Programming in Machiavelli - a Polymorphic Language with Static Type Inference". In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*. 1989, pages 46–57. DOI: 10.1145/67544.66931.

[54] Eyal Oren, Renaud Delbru, Sebastian Gerke, Armin Haller, and Stefan Decker. "ActiveRDF: object-oriented semantic web programming". In: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*. 2007, pages 817–824. DOI: 10.1145/1242572.1242682.

[55] Alexander Paar and Denny Vrandecic. "Zhi# - OWL Aware Compilation". In: *The Semanic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 - June 2, 2011, Proceedings, Part II*. 2011, pages 315–329. DOI: 10.1007/978-3-642-21064-8_22.

[56] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. "Practical pluggable types for java". In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*. 2008, pages 201–212. DOI: 10.1145/1390630.1390656.

[57] Fernando Silva Parreiras, Carsten Saathoff, Tobias Walter, Thomas Franz, and Steffen Staab. "APIs à gogo: Automatic Generation of Ontology APIs". In: *Proceedings of the 3rd IEEE International Conference on Semantic Computing (ICSC 2009), 14-16 September 2009, Berkeley, CA, USA*. 2009, pages 342–348. DOI: 10.1109/ICSC.2009.90.

[58] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8.

[59] Eric. Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. W3C Rec. https://www.w3.org/TR/rdf-sparql-query/. July 7, 2018. (Visited on 2018-07-07).

[60] Eclipse RDF4J. *RDF4J*. http://rdf4j.org/. (Visited on 2018-09-27).

[61] *RDFReactor*. http://semanticweb.org/wiki/RDFReactor. (Visited on 2018-09-27).

[62] Lukas Renggli. "Dynamic Language Embedding". PhD thesis. Institut für Informatik und angewandte Mathematik, Universität Bern, 2010.

[63] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. "Comparing libraries for generic programming in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. 2008, pages 111–122. DOI: 10.1145/1411286.1411301.

[64] Tim Sheard and Simon L. Peyton Jones. "Template meta-programming for Haskell". In: *SIGPLAN Notices* 37.12 (2002), pages 60–75. DOI: 10.1145/636517.636528.

[65] Lex Spoon and Seth Tisue. *Scala Compiler Plugins*. https://docs.scala-lang.org/overviews/plugins/index.html. 2018. (Visited on 2018-04-07).

[66] Timo Stegemann and Jürgen Ziegler. "Pattern-Based Analysis of SPARQL Queries from the LSQ Dataset". In: *Proceedings of the ISWC 2017 Posters & Demonstrations and Industry Tracks co-located with 16th International Semantic Web Conference (ISWC 2017), Vienna, Austria, October 23rd - to - 25th, 2017*. 2017. URL: http://ceur-ws.org/Vol-1963/paper542.pdf.

[67] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. "Secure distributed programming with value-dependent types". In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. 2011, pages 266–278. DOI: 10.1145/2034773.2034811.

[68] Don Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, Wonseok Chae, Uladzimir Matsveyeu, and Tomas Petricek. *F# 3.0 - Strongly-Typed Language Support for Internet-Scale Information Sources*. Technical report. Sept. 2012.

[69] Ben Szekely and Joe Betz. *Jastor*. http://jastor.sourceforge.net/. (Visited on 2018-09-27).

[70] Sesame Development Team. *Alibaba*. https://bitbucket.org/openrdf/alibaba. (Visited on 2018-09-27).

[71] Stardog Union. *Stardog*. https://www.stardog.com. (Visited on 2018-09-27).

[72] Denny Vrandecic and Markus Krötzsch. "Wikidata: a free collaborative knowledgebase". In: *Commun. ACM* 57.10 (2014), pages 78–85. DOI: 10.1145/2629489.

[73] W3C. *RDF 1.1 Concepts and Abstract Syntax*. https://www.w3.org/TR/rdf11-concepts/. July 7, 2018. (Visited on 2018-07-07).

[74] Limsoon Wong. "Kleisli, a Functional Query System". In: *J. Funct. Program.* 10.1 (Jan. 2000), pages 19–56. DOI: 10.1017/S0956796899003585.

A Full Metrics for UC 1

Table 4 Metrics for use case 1.

	`banana-rdf`	ScaSpa
Distinct operators n_1	13	11
Distinct operands n_2	29	16
Total operators N_1	53	19
Total operands N_2	54	24
Vocabulary n	42	27
Length N	107	43
Volume V	576.98	204.46
Difficulty D	12.10	8.25
Effort E	6983.42	1686.80
Compilation unit size s	27.7 KiB	9.8 KiB
Compilation time t_c	1.371 s	1.914 s (0.947 s)
Execution time t_r	0.060 s	0.063 s

B Full Metrics for UC 2

Table 5 Metrics for use case 2.

	`banana-rdf`	ScaSpa
Distinct operators n_1	19	10
Distinct operands n_2	31	14
Total operators N_1	51	19
Total operands N_2	54	21
Vocabulary n	50	24
Length N	105	40
Volume V	592.60	183.40
Difficulty D	16.55	7.50
Effort E	9806.66	1375.49
Compilation unit size s	25.7 KiB	9.2 KiB
Compilation time t_c	1.428 s	1.919 s (0.989 s)
Execution time t_r	0.265 s	0.317 s

C Listings for UC2

Listing 7 Use case 2 (`banana-rdf` implementation).

```
def professorsAndChairs(department: String): Try[Rdf#Solutions] = for {
   q <- parseSelect(
        s"""
          PREFIX : <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
          SELECT ?prof ?d WHERE {
            ?prof a :Professor .
            ?prof :worksFor <$department>.
            OPTIONAL { ?prof :headOf ?d . ?d a :Department }
          }""");
      r <- sparql.executeSelect(q)
  } yield r

professorsAndChairs("http://www.Department3.University0.edu") match {
  case Failure(error) => handleError(error)
  case Success(solution) =>
    solution.iterator.foreach { row =>
      val r = for {
        prof <- row("?prof");
        uriprof <- prof.as[Rdf#URI]
      } yield uriprof
      r match {
         case Failure(error) => handleError(error)
         case Success(uri) =>
          if (row("?d").isSuccess)
            println(s"$uri (CHAIR)")
          else
            println(uri)
      }
    }
}
```

Listing 8 Use case 2 (ScaSpa implementation).

```
def professors(department: `:Department`): List[`:Professor`] =
  sparql"""
    SELECT ?prof WHERE {
      ?prof a :Professor .
      ?prof :worksFor $department .
    }"""

professors(iri"http://www.Department3.University0.edu" : `:Department`)
  .foreach { prof =>
    prof match {
      case d: `:Chair` => println(s"$d (CHAIR)")
      case _ => println(prof)
    }
  }
```