Type Checking Program Code using SHACL

Martin Leinberger[1], Philipp Seifer[2], Claudia Schon[1], Ralf Lämmel[2], Steffen Staab[1,3]

[1] Institute for Web Science and Technologies, University of Koblenz-Landau, Germany
[2] The Software Languages Team, University of Koblenz-Landau, Germany
[3] Web and Internet Science Research Group, University of Southampton, England

Abstract. It is a strength of graph-based data formats, like RDF, that they are very flexible with representing data. To avoid run-time errors, program code that processes highly-flexible data representations exhibits the difficulty that it must always include the most general case, in which attributes might be set-valued or possibly not available. The Shapes Constraint Language (SHACL) has been devised to enforce constraints on otherwise random data structures. We present our approach, Type checking using SHACL (TyCuS), for type checking code that queries RDF data graphs validated by a SHACL shape graph. To this end, we derive SHACL shapes from queries and integrate data shapes and query shapes as types into a λ-calculus. We provide the formal underpinnings and a proof of type safety for TyCuS. A programmer can use our method in order to process RDF data with simplified, type checked code that will not encounter run-time errors (with usual exceptions as type checking cannot prevent accessing empty lists).

Keywords: SHACL · Programming with RDF · Type checking

1 Introduction

Graph-based data formats, such as RDF, have become increasingly popular, because they allow for much more flexibility for describing data items than rigidly-structured relational databases. Even when an ontology defines classes and properties, because of its open-world assumption, it is always possible to leave away required information or to add new classes and properties on the fly. Such flexibility incurs cost. Programmers cannot rely on structural restrictions of data relationships. For instance, the following T-Box axiom states that every `Student` has at least one `studiesAt` relation:

$$\texttt{Student} \sqsubseteq \geq 1\ \texttt{studiesAt}.\top \tag{1}$$

Consider an RDF data graph such as shown in Fig. 1. The two nodes `alice` and `bob` are both instances of `Student` and `Person`. For `alice`, only the name is known. For `bob`, name, age and that he studies at b_1, which is an instance of `University`. Such a graph is a valid A-Box for the T-Box stated above. However, for a program containing a variable x representing an instance of `Student`, there is no guarantee that the place of study is explicitly mentioned in the data and can be displayed. Depending on whether x contains `alice` or `bob`, the following program may succeed or encounter a run-time error:

```
print(x.studiesAt)
```

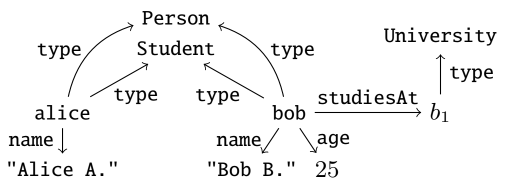

Fig. 1: Sample RDF data graph G_1.

The Shapes Constraint Language (SHACL) is a recent W3C recommendation [13] set out to allow for formulating integrity constraints. By now, a proposal for its formal semantics has been formulated by the research community [7] and SHACL shape graphs can be used to validate given data graphs. [13] itself states that:

> SHACL shape graphs [...] may be used for a variety of purposes besides validation, including user interface building, code generation and data integration.

However, it does not state *how* SHACL shape graphs might be used for these purposes. We consider the problem of writing code against an—possibly evolving—RDF data graph that is and remains conformant to a SHACL shape graph. We assume that the RDF database handles the rejection of transactions that invalidate conformance between SHACL shape graph and data graph. Then, the programming language should be able to type check programs that were written referring to a defined SHACL shape graph. Type checking should reject programs that could cause run-time errors, e.g., because they try to access an RDF property that is not guaranteed to exist without safety precautions. They should also simplify programs for which queries are guaranteed to return single values rather than lists, and they should accept programs that do not get stuck when querying conformant data graphs (with usual exceptions).

To exemplify this, consider three SHACL shapes `StudentShape`, `PersonShape` and `UniversityShape` (see Fig. 2). `StudentShape` validates all instances of `Student`, enforcing that there is at least one `studiesAt` relation, that all `studiesAt` relations point to a node conforming to the `UniversityShape` and that all instances of `Student` are also instances of `Person`. `PersonShape` validates all instances of `Person` and enforces the presence of exactly one `name` relation. `UniversityShape` enforces at least one incoming `studiesAt` relation and that all incoming `studiesAt` relations are from nodes conforming to the `StudentShape`. In order for G_1 to be valid with respect to the SHACL constraints above, either the statement that `alice` is an `Student` must be removed or a place of study for `alice` added. With these changes, the program above cannot fail anymore. A different program (see Lst. 1) may query for all instances of `Student`. The program may then try to access the `age` relation of each query result. However, since it is possible to construct an RDF graph that is validated by the shapes above, but lacks an `age` relation on some instances of `Student`, the program is unsafe and may crash with a run-time error. Contrary to that, a similar program that accesses the `name` relation instead is guaranteed to never cause run-time errors.

```
ex:StudentShape a sh:NodeShape;
  sh:targetClass ex:Student;
  sh:property [
    sh:path ex:studiesAt;
    sh:minCount 1;
    sh:node ex:UniversityShape ];
  sh:class ex:Person.

ex:PersonShape a sh:NodeShape;
    sh:targetClass ex:Person;
    sh:property [
      sh:path ex:name;
      sh:minCount 1;
      sh:maxCount 1 ].
ex:UniversityShape a
    sh:NodeShape;
  sh:property [
    sh:path [
      sh:inversePath;
      ex:studiesAt ];
    sh:minCount 1;
    sh:node
        ex:StudentShape ].
```

Fig. 2: SHACL constraints for RDF data graph G_1.

Listing 1: Program that may produce a run-time error.

```
map (fun x -> x.?X.age) (query {
    SELECT ?X WHERE { ?X rdf:type ex:Student.} })
```

Contributions We propose a type checking procedure based on SHACL shapes being used as types. We assume that a program queries an—possibly evolving—RDF data graph that is validated by a SHACL shape graph. Our contributions are then as follow:

1. We define how SHACL shapes can be inferred from queries. As queries are the main interaction between programs and RDF data graphs, inferring types from data access is a major step in deciding which operations are safe.
2. We then use a tiny core calculus that captures essential mechanisms to define a type system. Due to its simplicity, we use a simply typed λ-calculus whose basic model of computation is extended with queries. We define how SHACL shapes are used to verify the program through a type system and show that the resulting language is type-safe. That is, a program that passed type checking successfully does not yield run-time errors (with the usual exception of e.g., accessing the head of an empty list).

Organization The paper first recalls basic syntax and semantics for SPARQL and SHACL in Section 2. Then, the paper describes how we infer SHACL shapes from queries in Sections 3 and 4 before defining syntax and evaluation rules of the λ-calculus in Section 5. Then, the type system including subtyping is defined in Section 6 before showing its soundness in Section 7. Finally, we discuss related work in Section 8 and conclude in Section 9.

2 Preliminaries

2.1 SPARQL

RDF graphs are queried via the SPARQL standard [20]. We focus on a core fragment of SPARQL that features conjunctive queries (CQ) and simple path (P) expressions. We abbreviate this fragment by PCQ. That is, our queries are conjunctions of property path expressions that use variables only in place of graph nodes, not in place of path expressions[1] [3]. This is also a very widely used subset of SPARQL queries [18].

Syntax We denote the set of graph nodes of an RDF graph G by N_G with $v \in N_G$ denoting a graph node. Furthermore, we assume the existence of a set of variables N_V with x representing members of this set. The metavariable r denotes a SPARQL property path expression. A property path expression allows for defining paths of arbitrary length through an RDF graph. In our case, a property path is either a simple iri (i), the inverse of a path (r^-) or a path that connects subject to object via one or more occurrences of r (r^+). Lastly, we allow for path sequences (r_1/r_2). A PCQ $q = (\overline{x}) \leftarrow \mathit{body}$ consists of a head ($\overline{x}$) and a body. We use $\overline{x}$ to denote a sequence of variables $x_1, \ldots, x_n$. In a head of a PCQ ($\overline{x}$), the sequence $\overline{x}$ represents the answer variables of the query which are a subset of all variables occurring in the body of q. We use $\mathit{vars}(q)$ to refer to the set of all variables occurring in q. Fig. 3 summarizes the syntax.

q ::=	$(\overline{x}) \leftarrow \mathit{body}$	(query)
body ::=		$(\mathit{query\ body})$
	$\mathit{body} \wedge \mathit{body}$	(conjunction)
	$\mid \mathit{pattern}$	(pattern)
$\mathit{pattern}$ ::=		$(\mathit{pattern})$
	$x\ r\ v$	(subject var pattern)
	$\mid v\ r\ x$	(object var pattern)
	$\mid x\ r\ x$	(subject object var pattern)
r ::=	$i \mid r^- \mid r/r \mid r^+$	$(\mathit{path\ expresssions})$

Fig. 3: Syntax of PCQs.

Semantics For query evaluation, we follow standard semantics. Evaluation of a query over a graph G is denoted by $[\![\cdot]\!]_G$ and yields a set of mappings μ, mapping variables of the query onto graph nodes. The full evaluation rules can be found in the extended technical report of the paper[2].

[1] As we use plain RDF, we do not differentiate between distinguished and existential variables.
[2] Available on arxiv.org.

2.2 Shapes Constraint Language (SHACL)

The Shapes Constraint Language (SHACL) is a W3C standard for validating RDF graphs. In the following, we rely on the definitions presented by [7]. SHACL groups constraints in so-called *shapes*. A shape is referred to by a name, it has a set of constraints and defines its *target nodes*. Target nodes are those nodes of the graph that are expected to fulfill the constraints of the shape. As exemplified by `StudentShape` and `UniversityShape` (see Fig. 2), constraints may reference other shapes.

Constraint Syntax We start by defining constraints. We follow [7], who use a logical abstraction of the concrete SHACL language. Fragments of first order logic are used to simulate node shapes whereas so called property shapes are completely abstracted away. Constraints that are used in shapes are defined by the following grammar:

$$\phi ::= \top \mid s \mid v \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \geq_n r.\phi \tag{2}$$

where s is a shape name (indicating a reference to another shape), v is a constant (or rather a graph node), r is a property path and $n \in \mathbb{N}^+$. Additional syntactic constructs may be derived from this basic grammar, including $\leq_n r.\phi$ for $\neg(\geq_{n+1} r.\phi)$, $=_n r.\phi$ for $(\leq_n r.\phi) \wedge (\geq_n r.\phi)$ and $\phi_1 \vee \phi_2$ for $\neg(\neg\phi_1 \wedge \neg\phi_2)$. We sometimes use ϕ_s to denote the constraint belonging to a specific shape s. To improve readability, we sometimes add parenthesis to constraints although they are not explicitly mentioned in the grammar.

Constraint Evaluation Evaluation of constraints is rather straightforward with the exception of reference cycles. Evaluation is therefore grounded using assignments σ which map graph nodes to shape names [7]. We rely on total assignments instead of partial assignments for simplicity.

Definition 1 (Total Assignment). *Let G be an RDF data graph with its set of nodes N_G and let N_S a set of shape names. Then σ is a total function $\sigma : N_G \rightarrow 2^{N_S}$ mapping graph nodes v to subsets of N_S. If $s \in \sigma(v)$, then v is assigned to the shape s.*

Evaluation of a constraint ϕ for a node v of a graph G using an assignment σ is denoted $[\![\phi]\!]^{v,G,\sigma}$ and yields either true or false. The extended version contains the complete definition.

Shapes and Validation A shape is modelled as a triple (s, ϕ, q) consisting of a shape name s, a constraint ϕ and a query for target nodes q which is either an empty set or a monadic query that has exactly one answer variable to describe all intended *target nodes*. Target nodes denote those nodes which are expected to fulfill the constraint associated with the shape. In a slight abuse of notation, we write $v \in [\![q]\!]_G$ to indicate that a node v is a target node for s in the graph G. If S is a set of shapes, we assume that for each $(s, \phi, q) \in S$, if shape name s' appears in ϕ, then there also exists a $(s', \phi', q') \in S$. To illustrate this, consider our running example again (see Fig. 2)[3]. The

[3] We simplified target queries in the example—in reality, the target queries should query for Student or any of its subclasses. We simplified this as we do not use any RDFS subclass relations in our examples.

set S_1 containing all three shapes looks as follows:

$$\begin{aligned} S_1 = \{&(s_{Student}, \geq_1 \texttt{studiesAt}.\top \wedge \leq_0 \texttt{studiesAt}.\neg s_{University} \wedge \geq_1 \texttt{type.Person}, \\ &\quad (x_1) \leftarrow x_1\ \texttt{type Student}), \\ &(s_{Person}, =_1 \texttt{name}.\top, (x_1) \leftarrow x_1\ \texttt{type Person}), \\ &(s_{University}, \geq_1 \texttt{studiesAt}^-.\top \wedge \leq_0 \texttt{studiesAt}^-.s_{Student}, \emptyset)\} \end{aligned}$$

Intuitively, only certain assignments are of interest. Such an assignment is called a *faithful assignment.*

Definition 2 (Faithful assignment). *An assignment σ for a graph G and a set of shapes S is faithful, iff for each $(s, \phi, q) \in S$ and for each graph node $v \in N_G$, it holds that:*

– *if $v \in [\![q]\!]_G$, then $s \in \sigma(v)$.*
– *if $s \in \sigma(v)$, iff $[\![\phi]\!]^{v,G,\sigma} = true$.*

Validating an RDF graph means finding a faithful assignment. The graph is said to *conform* to the set of shapes.

Definition 3 (Conformance). *A graph G conforms to a set of shapes S iff there is a faithful assignment σ for G and S. We write $\sigma^{G,S}$ to denote that σ is a faithful assignment for G and S.*

Validating an RDF graph means finding a faithful assignment. In case of graph G_1 (see Fig. 1) and the set of shapes S_1, it is impossible to validate the graph. `alice` would need to be assigned to $s_{Student}$, but has no `studiesAt` relation. However, if the statement (`alice`,`type`,`Student`) is removed, then the graph is valid since a faithful assignment may assign s_{Person} to `alice` and `bob`, $s_{Student}$ solely to `bob` and $s_{University}$ to b_1.

3 Shape Inference for Queries

In this section, we describe how to infer shapes from PCQs for all variables in a given query. Given a query q with $x \in vars(q)$, let s^q_x be the globally unique shape name for variable x in query q. Then we assign the shape (s^q_x, ϕ, q_x). We discard sub- or superscripts if they are evident in context.

Our typing relation ":" for a PCQ q constructs a set of shapes S_q in the following manner: For every subject var pattern $x\ r\ v$ in the body of q (object var pattern $v\ r\ x$ respectively), we assign the constraint $\geq_1 r.v$ ($\geq_1 r^-.v$). As target nodes, we use the original query but projected on the particular variable. In case of variables on both subject and object ($x_1\ r\ x_2$), we infer two shapes $s^q_{x_1}$ and $s^q_{x_2}$. We use shape references to express the dependencies and infer the constraints $\geq_1 r.s^q_{x_2}$ and $\geq_1 r^-.s^q_{x_1}$. In case of a conjunction ($body_1 \wedge body_2$), we infer the sets of constraints for each query body individually and then combine the results using the operator $\boxtimes$. The relation $\boxtimes$ takes two sets of shapes S_{q_1} and S_{q_2} combines them into a unique set performing a full outer join on the shape names:

$$\begin{aligned} S_{q_1} \boxtimes S_{q_2} = &\{(s^q_{x_i}, \phi_i \wedge \phi_j, (x_i) \leftarrow body_i \wedge body_j) | (s^q_{x_i}, \phi_i, (x_i) \leftarrow body_i) \in S_{q_1} \\ &\wedge (s^q_{x_i}, \phi_j, (x_i) \leftarrow body_j) \in S_{q_2}\} \cup \\ &\{(s^q_{x_i}, \phi_i, q_i) | (s^q_{x_i}, \phi_i, q_i) \in S_{q_1} \wedge \neg\exists (s^q_{x_i}, \phi_j, q_j) \in S_{q_2}\} \cup \\ &\{(s^q_{x_j}, \phi_j, q_j) | \neg\exists (s^q_{x_j}, \phi_i, q_i) \in S_{q_1} \wedge (s^q_{x_j}, \phi_j, q_j) \in S_{q_2}\} \end{aligned}$$

Fig. 4 contains the complete set of rules for inferring sets of shapes from PCQs.

$$x\ r\ v : \{(s^q_x, \geq_1 r.v, (x) \leftarrow x\ r\ v)\} \text{ (R-SUB-VAR)}$$

$$v\ r\ x : \{(s^q_x, \geq_1 r^-.v, (x) \leftarrow v\ r\ x)\} \text{ (R-OBJ-VAR)}$$

$$x_1\ r\ x_2 : \{(s^q_{x_1}, \geq_1 r.s^q_{x_2}, (x_1) \leftarrow x_1\ r\ x_2), (s^q_{x_2}, \geq_1 r^-.s^q_{x_1}, (x_2) \leftarrow x_1\ r\ x_2)\} \text{ (R-VARS)}$$

$$\frac{body_1 : S_{q_1} \qquad body_2 : S_{q_2}}{body_1 \wedge body_2 : S_{q_1} \boxtimes S_{q_2}} \text{ (R-CONJ)} \qquad \frac{body : S_q}{(\overline{x}) \leftarrow body : S_q} \text{ (R-PROJ)}$$

Fig. 4: Inference rules for inferring a set of shapes from the body of query q.

As an example, consider the query $q = (x_1, x_2) \leftarrow x_1\ \texttt{type Student} \wedge x_1\ \texttt{studiesAt}\ x_2$ as used before. Then shape inference on the body assigns the following set of shapes:

$$\begin{aligned}
&(1)\ x_1\ \texttt{type Student}\ \wedge\ x_1\ \texttt{studiesAt}\ x_2 : (2) \boxtimes (3)\\
&\quad = \{(s^q_{x_1}, \geq_1 \texttt{type.Student} \wedge \texttt{studiesAt.}s^q_{x_2}, (x_1) \leftarrow x_1\ \texttt{type Student}\ \wedge\ x_1\ \texttt{studiesAt}\ x_2),\\
&\qquad (s^q_{x_2}, \geq_1 \texttt{studiesAt}^-.s^q_{x_1}, (x_2) \leftarrow x_1\ \texttt{type Student}\ \wedge\ x_1\ \texttt{studiesAt}\ x_2)\}\\
&(2)\ x_1\ \texttt{type Student} : \{(s^q_{x_1}, \geq_1 \texttt{type.Student}, (x_1) \leftarrow x_1\ \texttt{type Student})\}\\
&(3)\ x_1\ \texttt{studiesAt}\ x_2 : \{(s^q_{x_1}, \geq_1 \texttt{studiesAt.}s^q_{x_2}, (x_1) \leftarrow x_1\ \texttt{studiesAt}\ x_2),\\
&\qquad\qquad (s^q_{x_2}, \geq_1 \texttt{studiesAt}^-.s^q_{x_1}, (x_2) \leftarrow x_1\ \texttt{studiesAt}\ x_2)\}
\end{aligned}$$

4 Soundness of Shape Inference for Queries

Shape inference for queries is sound if the shape constraints inferred for each variable evaluate to true for all possible mappings of the variable.

Definition 4 (Soundness of shape inference). *Given an RDF graph G, a PCQ q with its variables $x_i \in vars(q)$ and the set of inferred shapes $S_q = \{(s^q_{x_i}, \phi_{x_i}, q_{s_{x_i}})^{x_i \in vars(q)}\}$, a shape constraint is sound if there exists a faithful assignment σ^{G,S_q} such that*

$$\forall x_i \in vars(q) : \forall \mu \in [\![q]\!]_G : [\![\phi_{x_i}]\!]^{\mu(x_i),G,\sigma^{G,S_q}} = true$$

We show that the faithful assignment σ^{G,S_q} can be constructed by assigning all shape names solely based on target nodes.

Theorem 1. *For any graph G, a PCQ q and the set of shapes S_q inferred from q, assignment σ^{G,S_q} is constructed such that for each shape $(s, \phi_s, q_s) \in S_q$ and for each graph node $v \in N_G$:*

1. *If $v \in [\![q_s]\!]_G$, then $s \in \sigma^{G,S_q}(v)$,*
2. *If $v \notin [\![q_s]\!]_G$, $s \notin \sigma^{G,S_q}(v)$.*

Such an assignment σ^{G,S_q} is faithful.

Proof (Sketch). Intuitively, a node v is part of the query result due to the presence of some relations for the node. The assigned constraints require the presence of the exact same relations to evaluate to true. A induction over the query evaluation rules can therefore show that 1) all nodes that are in the query result fulfill the constraint whereas 2) a node not being in the query result would also violate the constraint. ■

The faithful assignment σ^{G,S_q} constructed in the manner as explained above is unique. This is expected as shape inference does not use negation.

Proposition 1. *The assignment σ^{G,S_q} constructed as described above is unique.*

Proof. Assume that a different faithful assignment σ'^{G,S_q} exists. There must be at least one node v for which $\sigma^{G,S_q}(v) \neq \sigma'^{G,S_q}(v)$.

1. It is impossible that there is an s such that $s \in \sigma^{G,S_q}(v)$ and $s \notin \sigma'^{G,S_q}(v)$. σ assigns shapes based on target nodes, v must be a target node for s and σ' is not faithful.
2. It cannot be that $s \notin \sigma^{G,S_q}(v)$ and $s \in \sigma'^{G,S_q}(v)$. v must fulfill the constraint ϕ_s of shape s, otherwise σ' would not be faithful. If that is the case, then σ is not faithful. This contradicts Theorem 1. ■

Given a faithful assignment $\sigma^{G,S}$ for a set of shapes S and assignment σ^{G,S_q} for an inferred set of shapes, the two assignments can be combined by simply taking the union $\sigma^{G,S}(v) \cup \sigma^{G,S_q}(v)$ for each graph node $v \in N_G$. While not true for two arbitrary assignments, it is true in this case because shape names of S and S_q are disjoint.

5 Core Language

Syntax Our core language (Fig. 5) is a simply typed call-by-value λ-calculus. A program is a pair consisting of shapes written for the program S and a term. Terms (t) include[4] function application and if-then-else expressions. Constructs for lists are included in the language: **cons**, **nil**, **null**, **head** and **tail**. Specific to our language is a querying construct for querying an RDF graph with PCQs. To avoid confusion between PCQ query variables and program variables, we refer to the variables of a query always with the symbol l as they are treated as labels in the program. We assume labels to be either simple user-defined labels as commonly used in records, query variables or property paths. Labels are used for projection. In case of a projection for a record, the value associated with label is selected. When evaluating queries, evaluation rules turn query results into lists of records whereas answer variables are used as record labels. Lastly, in case of a projection for a graph node, the label is interpreted as a property path and the graph is traversed accordingly. Even though not explicitly mentioned in the syntax, we sometimes add parenthesis to terms for clarification. Values (val) include graph nodes, record values, nil and cons to represent lists, λ-abstractions and the two boolean values true and false. λ-abstractions indicate the type of their variable explicitly.

[4] Since they show no interesting effects, let statements and a fixpoint operator allowing for recursion, e.g., as necessary to define a `map` function are omitted. They are contained in the extended version.

P ::=	$(program)$
S, t	(program shapes and term)
t ::=	$(term)$
$t\ t$	(application)
∣ **if** t **then** t **else** t	(if-then-else)
∣ **cons** $t\ t$	(list constructor)
∣ **null** t	(test for empty list)
∣ **head** t	(head of list)
∣ **tail** t	(tail of list)
∣ **query** q	(query)
∣ $t.l$	(projection)
∣ $\{l_i = t_i^{i \in 1 \ldots n}\}$	(record)
∣ x	(variable)
∣ val	(value)

val ::=	$(values)$
v	(graph node)
∣ $\{l_i = val_i^{i \in 1 \ldots n}\}$	(record)
∣ **nil**$[T]$	(empty list)
∣ **cons** $val\ val$	(list constructor)
∣ $\lambda(x : T).t$	(abstraction)
∣ true	(true)
∣ false	(false)
T ::=	$(types)$
s	(shape name)
∣ $T \to T$	(function type)
∣ T list	(list type)
∣ $\{l_i : T_i^{i \in 1 \ldots n}\}$	(record type)
∣ bool	(boolean)
Γ ::=	$(context)$
$\emptyset$	(empty context)
∣ $\Gamma, x : T$	(type binding)

Fig. 5: Abstract syntax of λ_{SHACL}.

Types (T) include shape names (s) as well as type constructors for function ($T \to T$), list (T list) and record types ($\{l_i : T_i^{i \in 1 \ldots n}\}$). We assume primitive data types such as integers and strings, but omit routine details. To illustrate them, we include booleans in our syntax. As common in simply typed λ-calculi, we also require a context Γ for storing type bindings for λ-abstractions.

As an example, remember the program in Lst. 1 which queried for all instances of `Student`. Assuming that `map` is defined using basic recursion, the program can be expressed as

$$\textbf{map}\ (\lambda(y : \{x : s_{Student}\}).y.x.\texttt{age})\ (\textbf{query}\ (x_1) \leftarrow x\ \texttt{type Student})$$

In this program, the function (λ-abstraction) has one variable y whose type is a record. The record consists of a single label x, representing the answer variable of the query. The type of x is the shape $s_{Student}$. The term $y.x$ in the body of the function constitutes an access to the record label. Accessing the `age` in the next step constitutes a projection that traverses the graph. Type-checking rightfully rejects this program as nodes conforming to $s_{Student}$ may not have a `age` relation.

Semantics The operational semantics is defined using a reduction relation, which extends the standard ones. As types do not influence run-time behavior, shapes do not occur in the evaluation rules. However, we define the reduction rules with respect to an RDF graph G. Reduction of lists, records and other routine terms bear no significant differences from reduction rules as, e.g., defined in [19] (c.f. Fig 6, reduction rules for lists are only contained in the technical report). Reduction rules for queries and node projections are summarized by rules E-QUERY and E-PROJNODE in Fig. 6. A term

$$(G \Rightarrow S_P, t) \to (G \Rightarrow t) \text{ (E-PROGRAM)} \qquad \frac{G \Rightarrow t_1 \to t_1'}{G \Rightarrow t_1 t_2 \to G \Rightarrow t_1' t_2} \text{ (E-APP1)}$$

$$\frac{G \Rightarrow t_2 \to t_2'}{G \Rightarrow val_1 t_2 \to G \Rightarrow val_1 t_2'} \text{ (E-APP2)} \qquad \frac{G \Rightarrow t_1 \to G \Rightarrow t_1'}{G \Rightarrow t_1.l \to G \Rightarrow t_1'.l} \text{ (E-PROJ)}$$

$$G \Rightarrow (\lambda x : T.t_1) val_2 \to G \Rightarrow [x \mapsto val_2] t_1 \text{ (E-APPABS)}$$

$$G \Rightarrow \textbf{if } \text{true } \textbf{then } t_2 \textbf{ else } t_3 \to G \Rightarrow t_2 \text{ (E-IF-TRUE)}$$

$$G \Rightarrow \textbf{if } \text{false } \textbf{then } t_2 \textbf{ else } t_3 \to G \Rightarrow t_3 \text{ (E-IF-FALSE)}$$

$$\frac{G \Rightarrow t_1 \to t_1'}{G \Rightarrow \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \to G \Rightarrow \textbf{if } t_1' \textbf{ then } t_2 \textbf{ else } t_3} \text{ (E-IF)}$$

$$\frac{G \Rightarrow t_j \to t_j'}{\begin{array}{c} G \Rightarrow \{l_i = val_i^{i \in 1 \ldots, j-1}, l_j = t_j, l_k = t_k^{k \in j+1 \ldots n}\} \to \\ G \Rightarrow \{l_i = val_i^{i \in 1 \ldots, j-1}, l_j = t_j', l_k = t_k^{k \in j+1 \ldots n}\} \end{array}} \text{ (E-RCD)}$$

$$G \Rightarrow \{l_i = val_i^{i \in 1 \ldots n}\}.l_j \to G \Rightarrow val_j \text{ (E-PROJRCD)}$$

$$\frac{q = (l_1, \ldots, l_n) \leftarrow body \qquad [\![q]\!]_G = \{\mu_1, \ldots, \mu_m\}}{\begin{array}{c} (G \Rightarrow \textbf{query } q) \to G \Rightarrow \textbf{cons } \{l_i = \mu_1(l_i)^{i \in 1, \ldots, n}\}, \ldots, \\ \textbf{cons } \{l_i = \mu_m(l_i)^{i \in 1, \ldots, n}\}, \textbf{nil} \end{array}} \text{ (E-QUERY)}$$

$$\frac{[\![(x) \leftarrow l(v, x)]\!]_G = \{\mu_1, \ldots, \mu_n\}}{G \Rightarrow v.l \to G \Rightarrow \textbf{cons } \mu_1(x) \ldots \textbf{cons } \mu_n(x) \textbf{ nil}} \text{(E-PROJNODE)}$$

Fig. 6: Reduction rules of λ_{SHACL}.

representing a query can be directly evaluated to a list of records. Query evaluation $[\![q]\!]_G$ returns a list of mappings. As in other approaches (e.g., [2]), each query result becomes a record of the list. For each record, labels are created for each variable whereas the value of the record is the value provided by the mapping. A projection on a given graph node is evaluated as a query by turning the property path expression l into a query pattern. However, instead of a record a plain list of graph nodes is returned.

Any term t which cannot be reduced any further (i.e. no rule applies to the term anymore) is said to be in *normal form*. When evaluation is successful, then the term has been reduced to a value val. Any term that is in normal form but not a value is said to be stuck. As usual [19], we use “stuckness” as a simple notion of a run-time error.

6 Type system

The most distinguishing feature of the type system is the addition of shape names as types in the language. As each shape name requires a proper definition, our typing relation ":" is defined with respect to a set of shapes. Likewise, a typing context Γ is required to store type bindings for λ-abstractions. Since certain constructs such as queries create new shapes during the type checking process, the typing relation does not only assign a type to a term but also a set of newly created shapes which in turn may contain definitions of shape names that are being used as types.

For the typing rules, we require the definition function lub that computes the least upper bound of two types. The exact definition can be found in the technical report. Intuitively, in case of two shapes s_1 and s_2, we rely on disjunction $s_1 \vee s_2$ as a least upper bound.

Typing rules The typing rules for constructs unrelated to querying are mainly the standard ones as common in simply typed λ-calculi, except all rules are defined with respect to a set of shapes and return a set of newly created shapes (see Fig. 7). Basic rules, such as for boolean values (rules T-TRUE and T-FALSE) simply return empty sets of shapes as they do not create new shapes. Several rules take possible extensions of the set of shapes into account. E.g., rule T-PROGRAM takes the set of shapes as defined by the program S_P and the pre-defined set of shapes S and uses the union of both to analyze the term t.

New shapes are mainly created when either the least upper bound judgement is used or one of the two query expressions (either **query** or projections) are used (see rules T-QUERY and T-NPROJ in Fig. 7). In case of a **query** statement (rule T-QUERY), the shape inference rules as described in Section 3 are being used to construct the set S_q which is being returned as newly created shapes. The actual type of a query then comprises a list of records. Each record contains one label per answer variable whereas the type of each label is the respective shape name for the query variable. Likewise, projections on graph nodes (T-NODEPROJ) create a new shape name s' using a function $genName$ based on the old shape name s with the appropriate constraint $\geq_1 l^{-}.s$. The newly created definition is returned as a set with the actual type of the expression being s list.

Subtyping Subtyping rules are summarized in Fig. 8. We rely on a standard subtyping relation. A term t of type T_1 is also of type T_2, if $T_1 <: T_2$ is true (T-SUB). Any type is always a subtype of itself (S-RELF). If T_1 is a subtype of T_2 and T_2 is a subtype of T_3, then T_1 is also a subtype of T_3 (S-TRANS). Subtyping for lists and functions is reduced to subtyping checks for their associated types. A list T_1 list is a subtype of T_2 list if T_1 is a subtype of T_2 (S-LIST). Function types are in a subtyping relation (S-FUNC) if their domains are in a flipped subtyping relationship ("contra-variance") and their co-domains are in a subtyping relationship ("co-variance"). Record type is a subtype of another record if 1) it has the the same plus more fields (S-RCDWIDTH), 2) it is a permutation of the supertype (S-RCDPERM) and 3) if the types of the fields are in a subtype relation (S-RCDDEPTH).

$$\frac{S \cup S_P, \Gamma \vdash t_1 : T_1, S_1}{S, \Gamma \vdash S_P, t_1 : T_1, S_1} \text{ (T-PROGRAM)}$$

$$\frac{S, \Gamma \vdash t_1 : T_{11} \to T_{12}, S_1 \qquad S, \Gamma \vdash t_2 : T_{11}, S_2}{S, \Gamma \vdash t_1 t_2 : T_{12}, S_1 \cup S_2} \text{ (T-APP)}$$

$$\frac{\begin{array}{c} S, \Gamma \vdash t_1 : \text{bool}, S_1 \\ S, \Gamma \vdash t_2 : T_2, S_2 \qquad S, \Gamma \vdash t_3 : T_3, S_3 \qquad lub(T_2, T_3, S \cup S_2 \cup S_3) = T_{lub}, S_{lub} \end{array}}{S, \Gamma \vdash \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 : T_{lub}, S_1 \cup S_2 \cup S_3 \cup S_{lub}} \text{ (T-IF)}$$

$$S, \Gamma \vdash \textbf{nil}[T] : T \text{ list}, \emptyset \text{ (T-NIL)} \qquad \frac{S, \Gamma \vdash t_1 : T \text{ list}, S_1}{S, \Gamma \vdash \textbf{tail } t_1 : T \text{ list}, S_1} \text{ (T-TAIL)}$$

$$\frac{S, \Gamma \vdash t_1 : T_1, S_1 \qquad S, \Gamma \vdash t_2 : T_1 \text{ list}, S_2}{S, \Gamma \vdash \textbf{cons } t_1\ t_2 : T_1 \text{ list}, S_1 \cup S_2} \text{ (T-CONS)}$$

$$\frac{S, (\Gamma, x : T_1) \vdash t : T_2, S_2}{S, \Gamma \vdash \lambda(x : T_1).t : T_1 \to T_2, S_2} \text{ (T-ABS)} \qquad \frac{x : T \in \Gamma}{S, \Gamma \vdash x : T, \emptyset} \text{ (T-VAR)}$$

$$S, \Gamma \vdash \text{true} : \text{bool}, \emptyset \text{ (T-TRUE)} \qquad S, \Gamma \vdash \text{false} : \text{bool}, \emptyset \text{ (T-FALSE)}$$

$$\frac{S, \Gamma \vdash t_1 : T_1 \text{ list}, S_1}{S, \Gamma \vdash \textbf{null } t_1 : \text{bool}, S_1} \text{ (T-NULL)} \qquad \frac{S, \Gamma \vdash t_1 : T_1 \text{ list}, S_1}{S, \Gamma \vdash \textbf{head } t_1 : \text{T}, S_1} \text{ (T-HEAD)}$$

$$\frac{\text{for each } i \qquad S, \Gamma \vdash t_i : T_i, S_i}{S, \Gamma \vdash \{l_i = t_i^{1 \in 1 \ldots n}\} : \{l_i : T_i^{i \in 1 \ldots n}\}, \bigcup_{i=1}^{n} S_i} \text{ (T-RCD)}$$

$$\frac{S, \Gamma \vdash t_1 : \{l_i : T_i^{i \in 1, \ldots, n}\}, S_1}{S, \Gamma \vdash t_1.l_i : T_i, S_1} \text{ (T-RCDPROJ)}$$

$$\frac{\begin{array}{c} q = (l_1, \ldots, l_n) \leftarrow body \\ vars(q) = \{l_1, \ldots, l_n, \ldots l_m\} \qquad q : S_q = \{(s^q_{l_i}, \phi^q_{l_i}, q^q_{s_{l_i}})^{i \in 1 \ldots m}\} \end{array}}{S, \Gamma \vdash \textbf{query } q : \{(l_i : s^q_{l_i})^{i \in 1 \ldots n}\} \text{ list}, S_q} \text{ (T-QUERY)}$$

$$\frac{S, \Gamma \vdash t_1 : s, S_1 \qquad genName(s) = s' \qquad S \cup \{s', \geq_1 l^-.s, \emptyset\} \vdash s <: s'}{S, \Gamma \vdash t_1.l : s' \text{ list}, S_1 \cup \{s', \geq_1 l^-.s, \emptyset\}} \text{ (T-NPROJ)}$$

Fig. 7: Typing rules for λ_{SHACL}.

Subtyping relations between two shapes s_1 and s_2 are defined via faithful assignments. An assignment $\sigma : N_G \to 2^{N_S}$ is a function that assigns shape names to graph nodes. We require the opposite direction—a function σ_{inv} assigning nodes to shapes.

Definition 5 (Inverse assignments). *Let G be an RDF data graph, S a set of shapes and $\sigma^{G,S}$ a faithful assignment for G and S. Then $\sigma_{inv}^{G,S}$ is a total function $\sigma_{inv}^{G,S} : N_S \to 2^{N_G}$ mapping shape names to subsets of N_G such that for all graph nodes $v \in N_G$ and all shape names $s \in N_S$: $s \in \sigma^{G,S}(v)$ iff $v \in \sigma_{inv}^{G,S}(s)$*

For a given set of shapes S, two shapes s_1 and s_2 are in a subtyping relation if, for all possible RDF graphs $G \in \mathcal{G}$ and all faithful assignments $\Sigma^{G,S}$ for S and G, it holds that $\sigma_{G,S}^{inv}(s_1) \subseteq \sigma_{G,S}^{inv}(s_2)$ (S-SHAPE). That is, the sets of nodes conforming to the two shapes are in a subset relation for all possible RDF graphs conform to the set of shapes.

$$\frac{S, \Gamma \vdash t_1 : T_1, S_1 \qquad S \vdash T_1 <: T_2}{S, \Gamma \vdash t_1 : T_2, S_1} \text{ (T-SUB)} \qquad S \vdash T <: T \text{ (S-REFL)}$$

$$\frac{S \vdash T_1 <: T_2 \qquad S \vdash T_2 <: T_3}{S \vdash T_1 <: T_3} \text{ (S-TRANS)}$$

$$\frac{S \vdash T_{21} <: T_{11} \qquad S \vdash T_{12} <: T_{22}}{S \vdash T_{11} \to T_{12} <: T_{21} \to T_{22}} \text{ (S-FUNC)} \qquad \frac{S \vdash T_1 <: T_2}{S \vdash T_1 \text{ list} <: T_2 \text{ list}} \text{ (S-LIST)}$$

$$S \vdash \{l_i : T_i^{i \in 1 \ldots n+k}\} <: \{l_i : T_i^{i \in 1 \ldots n}\} \text{ (S-RCDWIDTH)}$$

$$\frac{\{k_j : T_j^{j \in 1 \ldots n}\} \text{ is a permutation of } \{l_i : T_i^{i \in 1 \ldots n}\}}{S \vdash \{k_j : T_j^{j \in 1 \ldots n}\} <: \{l_i : T_i^{i \in 1 \ldots n}\}} \text{ (S-RCDPERM)}$$

$$\frac{\text{for each } i \qquad T_i <: T_i'}{S \vdash \{l_i : T_i^{i \in 1 \ldots n}\} <: \{l_i : T_i'^{i \in 1 \ldots n}\}} \text{ (S-RCDDEPTH)}$$

$$\frac{\forall G \in \mathcal{G} : \forall \sigma_{G,S}^{inv} \in \Sigma_{G,S}^{inv} : \sigma_{G,S}^{inv}(s_1) \subseteq \sigma_{G,S}^{inv}(s_2)}{S \vdash s_1 <: s_2} \text{ (S-SHAPE)}$$

Fig. 8: Subtyping rules.

Algorithmic subtyping Algorithmic solutions to standard subtyping rules such used in Fig. 8 are, e.g., described by [19]. In the case of subtyping for shapes, algorithmic approaches similar to subsumption checking in description logics [1] can be employed. That is, s_1 must be a subtype of s_2 if it can be shown that no graph exists that contains a node v for which $s_1 \in \sigma^{G,S}(v)$ but $s_2 \notin \sigma^{G,S}(v)$. As of now, we compare constraint sets which is sound but incomplete. We don't know whether a complete algorithm exists, although we plan to investigate a transformation into a description logic based reasoning problem.

Type elaboration Types do not play any role during the evaluation of terms. They are only used during the type checking process. This is by design, as run-time type checks

$$\ldots \qquad \frac{S, \Gamma \vdash t_1 : t_1', T_1 \text{ list}, S_1}{S, \Gamma \vdash \textbf{head}\ t_1 : \textbf{head}\ t_1', T_1, S_1} \text{ (T-HEAD)}$$

$$\frac{\begin{array}{c} S, \Gamma \vdash t_1 : t_1', s, S_1 \\ S \cup S' \cup \{s_{tmp}, =_1 l.\top, \emptyset\} \vdash s <: s_{tmp} \qquad genName(s) = s' \end{array}}{S, \Gamma \vdash t_1.l : \textbf{head}\ t_1'.l, s', S_1 \cup \{s', \geq_1 l^{-}.s, \emptyset\}} \text{ (T-NPROJ-1)}$$

$$\frac{\begin{array}{c} S, \Gamma \vdash t_1 : t_1', s, S_1 \qquad S \cup S' \cup \{s_{tmp}, =_1 l.\top, \emptyset\} \nvdash s <: s_{tmp} \\ S \cup S' \cup \{s_{tmp}, \geq_1 l.\top, \emptyset\} \vdash s <: s_{tmp} \qquad genName(s) = s' \end{array}}{S, \Gamma \vdash t_1.l : t_1'.l, s' \text{ list}, \{s', \geq_1 l^{-}.s, \emptyset\}} \text{ (T-NPROJ-2)}$$

Fig. 9: Type system with type elaboration (excerpt).

incur overhead and should be avoided, in particular if the type check is computationally expensive. However, the evaluation relation only evaluates terms of the form $v.l$ (node projections) into lists of graph nodes (c.f. rule E-PROJNODE of Fig. 6 and T-NPROJ of Fig. 7), even though a shape may hint that there is only one successor (e.g., `studiesAt` of shape $s_{Student}$). As the evaluation rules have no information about types, the type system must annotate or transform terms such that they can be treated differently during run-time. This process is called *type elaboration* [19]. The typing relation ":" then takes a set of shapes S and a typing context Γ and returns a term t, a type T and a set of newly introduced shapes S'. This is exemplified by the rules in Fig. 9. Most rules simply return the term without modifications (e.g., rule T-HEAD). However, in case of node projections where it can be shown that there is only a single successor, a **head** is automatically added to the term (rule T-NPROJ-1). Otherwise, the term is not modified (rule T-NPROJ-2).

7 Type Soundness

A term t is said to be well-typed if the type system assigns a type. We show the soundness of the λ_{SHACL} type system by proving that a well-typed term does not get stuck during evaluation. As with other languages, there are exceptions to this rule, e.g., downcasting in object-oriented languages, c.f. [10]. For λ_{SHACL}, this exception concerns lists. We show that if a program is well-typed, then the only way it can get stuck is by reaching a point where it tries to compute **head nil** or **tail nil**. Furthermore, terms must be closed, meaning that all program variables are bound by function abstractions [19]. We proceed in two steps, by showing that a well-typed term is either a value or it can take a step (progress) and by showing that if that term takes a step, the result is also well-typed (preservation).

Lemma 1 (Canonical Forms Lemma). *Let val be a well-typed value. Then the following observations can be made:*

1. *If val is a value of type s, then val is of the form v.*

2. *If val is value of type $T_1 \to T_2$, then val is of the form $\lambda(x : T_1).t_2$.*
3. *If val is a value of type T list, then val is either of the form* ***cons*** *val . . . or* ***nil****.*
4. *If val is a value of type $\{l_i : T_i^{i\in1...n}\}$, then val is of the form $\{l_i = val_i^{i\in1...n}\}$.*
5. *If val is a value of type bool, then val is either of the form true or false.*

Given Lemma 1, we can show that a well-typed term is either a value or it can take a step.

Theorem 2 (Progress). *Let t be a closed, well-typed term. If t is not a value, then there exists a term t' such that $t \to t'$. If $S, \Gamma \vdash t : T, S'$, then t is either a value, a term containing the forms* ***head nil*** *or* ***tail nil****, or there is some t' with $t \to t'$.*

Proof (Sketch). The theorem can be shown by induction on the derivation of $S, \Gamma \vdash t : T, S$. Queries ($t =$ **query** q) are straightforward as no sub-term exists. For node projections ($t_1.l$ with the type of t_1 being a shape name), Lemma 1 tells us that it must ultimately reduce to a graph node. In that case rule E-PROJNODE applies. The full proof can be found in the tech report. ■

Given that a well-typed term can take a step, we now need to show that taking a step according to the evaluation rules preserves the type.

Theorem 3 (Preservation). *Let t be a term and T a type. If $S, \Gamma \vdash t : T, S'$ and $t \to t'$, then $S, \Gamma \vdash t' : T, S'$.*

Proof (Sketch). As with progress, the proof is an induction over the typing relation $S, \Gamma \vdash t : T, S'$. For each term, possible ways of reducing it are distinguished and it is shown that in each case the type does not change. For queries, this is immediate. In case of node projections, t_1 either took a step, in which case the typing rule applies again, or it is a graph node v with type s. Each v' which is reached via the node projection conforms to the newly created shape s' with its constraint $\geq_1 l^-.s$. Therefore, the type is also preserved. ■

As a direct consequence of Theorems 2 and 3, a well-typed, closed term does not get stuck during evaluation.

8 Related Work

The presented approach is generally related to the validation of RDF as well as the integration of RDF into programming languages. RDF validation has seen an increase in interest. Among them are inference-based approaches such as [23,16], in which OWL expressions are used as integrity constraints by relying on a closed-world assumption. The fact that constraints are OWL expressions puts these approaches closer to [15] than the approach described here. A validation approach that is relatively similar to SHACL is ShEx [4]. ShEx also uses shapes to group constraints, but removes property path expressions and features well-defined recursion. We chose SHACL over ShEx due to SHACL being a W3C recommendation. Due to the similarity between SHACL and ShEx, the integration process for the latter is very similar. In fact, the definition for recursion used in ShEx even simplifies some aspects as there is no need for the notion of faithful assignments.

In terms of integration of RDF into programming languages, we consider different approaches. Generic representations, e.g., the OWL API [9] or Jena [5], use types on a meta-level (e.g., *Statement*) that do not allow a static type-checker to verify a program. This leaves correctness entirely on the hands of the programmer. Mapping approaches use schematic information of the data model to create types in the target language. Type checking can offer some degree of verification. An early example of this is OWL2Java [12], a more recent one is LITEQ [14]. However, mapping approaches based on ontologies come with their own limitations. OWL relies on a open-world assumption, in which missing information is treated as incomplete data rather than constraint violations. As shown in the introduction, structural information does therefore not necessarily imply the presence of data relationships. This is problematic for type-checkers as they rely on a closed world. The most powerful approaches create new languages or extend existing ones to accomodate the specific requirements of the data model. Examples include rule-based programming [11] as well as a transformation and validation language [21]. However, both are untyped. Typed approaches to linked data is provided by [8,6]. Zhi# [17], an extension of the C# language provides an integration for OWL ontologies, albeit it only considers explicitly given statements. Contrary to that, [15,22] provides an integration of OWL ontologies also considering implicit statements. However, as shown in the introduction, programmers cannot rely on structural restrictions given by OWL ontologies whereas SHACL enforces its structural restriction with a closed-world assumption.

9 Summary and Future Work

In this paper, we have presented an approach for type checking programs using SHACL. We have shown that by using SHACL shapes as types, type safety can be achieved. This helps in writing less error-prone programs, in particular when facing evolving RDF graphs. The work can be extended in several directions.

First, an implementation of the presented approach is highly desirable. Comparably to [22], we plan on implementing the approach in Scala using compiler plugins that add new compilation phases. Shape names constitute a new form of types. As shape names are known before compilation, they can be syntactically integrated using automatically generated type aliases to a base type. This allows for type checking shape types in a separate compilation phase that runs after the standard Scala type inference and type checker phases. As there is little interaction between normal Scala types and shape types, issues only arise when code converts e.g., literals into standard Scala types. However, this can be solved through minor code transformations before the type checking phase. Lastly, transformations based on type elaboration can also run as a separate phase. As shape types do not influence run-time behavior, compilation produces standard JVM byte code. However, one noteworthy limitation of using type aliases to represent shape names is that method overloading based on shape names is not possible. Resolving this issue requires better integration techniques which remain as future work.

Second, finding sound and complete methods for deciding shape subsumption is an interesting problem that requires future research. This is an important step as it defines practical boundaries in terms of the parts of SHACL that can be used for type checking. Lastly, the supported subset of SPARQL queries is relatively small and should be extended by missing features such as union of queries or filter expressions. This raises questions about the parts of SPARQL that can be described with SHACL shapes.

References

1. Baader, F., et al. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
2. Bierman, G.M., et al.: The Essence of Data Access in C*omega*. In: Proc. ECOOP 2005. pp. 287–311 (2005)
3. Bischof, S., et al.: Schema-Agnostic Query Rewriting in SPARQL 1.1. In: Proc. ISWC 2014. pp. 584–600. LNCS, Springer (2014)
4. Boneva, I., et al.: Semantics and Validation of Shapes Schemas for RDF. In: Proc. ISWC 2017. LNCS, vol. 10587, pp. 104–120. Springer (2017)
5. Carroll, J.J., et al.: Jena: implementing the semantic web recommendations. In: Proc. WWW 2004. pp. 74–83. ACM (2004)
6. Ciobanu, G., et al.: Minimal type inference for Linked Data consumers. J. Log. Algebr. Meth. Program. **84**(4), 485–504 (2015)
7. Corman, J., et al.: Semantics and Validation of Recursive SHACL. In: Proc. ISWC 2018. pp. 318–336. LNCS, Springer (2018)
8. Horne, R., et al.: A verified algebra for read-write Linked Data. Science of Computer Programming **89, Part A**, 2 – 22 (2014)
9. Horridge, M., et al.: The OWL API: A Java API for OWL ontologies. Semantic Web **2**(1), 11–21 (2011)
10. Igarashi, A., et al.: Featherweight Java: A Minimal Core Calculus for Java and GJ. ACM Transactions on Programming Languages and Systems **23**(3), 396–450 (May 2001)
11. Käfer, T., et al.: Rule-based Programming of User Agents for Linked Data. In: Proc. Linked Data on the Web. CEUR Workshop Proceedings, CEUR-WS.org (2018)
12. Kalyanpur, A., et al.: Automatic Mapping of OWL Ontologies into Java. In: Proc. Software Engineering & Knowledge Engineering (SEKE) 2004. pp. 98–103 (2004)
13. Knublauch, H., et al.: Shapes Constraint Language (SHACL). W3C Recommendation (2017), https://www.w3.org/TR/shacl/
14. Leinberger, M., et al.: Semantic Web Application Development with LITEQ. In: Proc. ISWC. pp. 212–227. LNCS, Springer (2014)
15. Leinberger, M., et al.: The Essence of Functional Programming on Semantic Data. In: Proc. European Symp. on Programming. pp. 750–776. LNCS, Springer (2017)
16. Motik, B., et al.: Adding Integrity Constraints to OWL. In: Proc. OWLED 2007. CEUR Workshop Proceedings, vol. 258. CEUR-WS.org (2007)
17. Paar, A., et al.: Zhi# - OWL Aware Compilation. In: Proc. of ESWC. pp. 315–329. LNCS, Springer (2011)
18. Picalausa, F., et al.: A Structural Approach to Indexing Triples. In: Proc. ESWC 2012. pp. 406–421. LNCS, Springer (2012)
19. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
20. Prud'hommeaux, E., et al.: SPARQL Query Language for RDF. W3C Rec. (Nov 2013), https://www.w3.org/TR/rdf-sparql-query/
21. Prud'hommeaux, E., et al.: Shape expressions: an RDF validation and transformation language. In: Proc. SEMANTICS 2014. pp. 32–40. ACM (2014)
22. Seifer, P., et al.: Semantic Query Integration With Reason. Programming Journal **3**(3), 13 (2019)
23. Tao, J., et al.: Integrity Constraints in OWL. In: Proc. AAAI 2010. AAAI Press (2010)