DRAFT!
On typing knowledge graphs

Iztok Savnik[1]

Department of computer science,
Faculty of mathematics, natural sciences and information technologies,
University of Primorska, Slovenia
iztok.savnik@upr.si

Abstract. The problem addressed in this paper is typing a single ground triple t from a knowledge graph. First, the *ground type* of t is inferred bottom-up, from the stored typing of individual entities which are the components of t. The ground type is further minimized and then generalized to obtain a minimal upper bound (MUB) type. The MUB type is an appropriate starting point for exploring its relations to the conceptual schema of a knowledge graph. Second, the *schema type* of t is inferred from the conceptual schema based on the predicate of t. All valid schema types of t are gathered and then minimized to obtain a schema type with a minimal interpretation. Finally, the minimal schema type is filtered by relating it to the MUB ground type via the subtype relation to compute the final schema type of t.

Keywords: knowledge representation · knowledge graphs · type systems .

1 Introduction

A knowledge graph (abbr.. KG) includes a data and knowledge representation language in the form of a graph. It is the dictionaries, such as RDF-Schema [25], which attach meanings to the edges of a graph, turning a graph into a language for the representation of data and knowledge. In comparison to the data and program structures used in programming languages [22, 10], the conceptual schema of a KG is more expressive: in principle, any data and program structure can be represented as a graph. The expressive power comes from logic that stands behind a KG, where a graph is a set of named relations, and triples are logical statements. Let us depict some of the features of the data and knowledge representation language of a KG.

First, a KG includes predefined classes and predicates that are organized into taxonomies [2]. Second, typing of ground identifiers, as well as typing of the predicates with the domain and range types, is stored in a KG. Further, similarly to the *roles* [4] of a knowledge base, the predicates of a KG are treated as classes (types) that are included in a taxonomy of predicates. However, they also act as individual entities described with additional predicates. Finally, the predicates are inherited through the taxonomy of classes, and, from the other point of view, the domain and range types of the predicates are inherited through the hierarchy of predicates.

Typing a knowledge graph requires a framework for the definition of rules, which is different from the framework for classical typing of programming languages. While typing in programming languages is based on the syntactical composition of a program, a knowledge graph is built from simple triples. Triples can form rich semantic structures depending on the kinds of entities and values. The kind of object, for example, can be either a ground value or a class. Typing of a KG must, therefore, be able to grasp the kinds of objects in the rules, and, in many cases, need to be guided by a procedure.

From a logic perspective, the ground entities and triples represent an ABox, while the schema of KG is a TBox and represents logical statements about classes and binary relations. The syntactical structures that we use for types are the following. First, we have the classes and predicates, the types of identifiers, that are identifiers themselves. Second, the type of triples is a product type, a triple itself. Next, the $\wedge$ and $\vee$ types capture many requirements of the KG domain very naturally. For instance, entities belong to multiple classes simultaneously, or a domain of a predicate is two different classes. Finally, the taxonomies of classes and predicates are hard to grasp structurally; rather, posets can be manipulated with algebraic operations.

The paper presents a study of typing a single ground triple t. We propose to infer first the ground type of t using the stored typing of identifiers. The ground type is minimized and then generalized to the minimal upper bound (MUB) type. The MUB type is an appropriate starting point to search for the triple types from a KG conceptual schema.

The final triple type of $t = (s, p, o)$ is selected from the triple types provided by a conceptual schema stored in a KG. Using the predicate p as the starting point, we choose first all valid triple types including a predicate p or one of p's super-predicates. The set of selected triple types is minimized to obtain a set of minimal and unrelated triple types, the candidates for the type of t.

The appropriate candidate triple types are selected by relating the candidates to the MUB ground type by means of a subtype relation. The disjunctively bound candidate triple types that are related to the MUB ground type via subtype relation form the final type of t.

The contributions of the presented research are as follows. First, to our knowledge, this is the first proposal for typing a knowledge graph that encompasses a complete KG. The existing approaches to typing a KG deal with the particular problems of typing a KG. Second, the presented framework for typing ground triples can serve conveniently for the implementation of type-checking the stored typing of a KG. The analysis of typing an individual triple spans from the ground types at the lower levels of ontology to the minimal schema types at the upper levels of ontology. Finally, we show that the triple types can be used to disambiguate the sense of a predicate with multiple meanings.

...

2 Preliminaries

2.1 Knowledge graph

This section defines a knowledge graph as a RDF graph [24] using RDF-Schema [25] for the representation of the structural part of a knowledge base.

Let I be a set of URI-s, B be a set of blanks and L be a set of literals. Let us also define sets $S = I \cup B$, $P = I$, and $O = I \cup B \cup L$. A *RDF triple* is a triple $(s, p, o) \in S \times P \times O$. A *RDF graph* $g \subseteq S \times P \times O$ is a set of triples. Set of all graphs will be denoted as G.

The complete set of triples of a RDF graph is in the text denoted as $\varDelta$. We use $\varDelta$ when we want to refer to the original set of KG triples and not the data model of a KG presented in the following Section.

2.2 A data model of a KG

To abstract away the details of the RDF data model we unify the representation of knowledge graph by separating solely between the identifiers and triples. The identifiers include the set of individual identifiers and the set of class identifiers. The triples inlcude the set of ground triples and the set of triple types.

However, since typing of a KG is based on the separation between the values and the types, we use the following classification of identifiers and triples. The set of values $\mathcal{V}$ includes the individual identifiers $\mathcal{V}_i$ and the individual (ground) triples $\mathcal{V}_t$. To be able to refer to the specific subsets of $\mathcal{V}_i$ in the rules, we also introduce the set $\mathcal{V}_l \subseteq \mathcal{V}_i$, which denotes a set of literal values, and the set $\mathcal{V}_p \subseteq \mathcal{V}_i$ that refers to the set of predicates from a KG. Finally, a set $\mathcal{T}_p \in \mathcal{T}_i$ stands for all predicates of a KG that now have the role of types. Note that predicates are treated both as values and as types.

The set of all valid types $\mathcal{T}$ of a KG comprises the class identifiers $\mathcal{T}_i$, i.e. types of individual identifiers $\mathcal{V}_i$, and the triple types $\mathcal{T}_t$, i.e. the types of individual triples $\mathcal{V}_t$.

The set of class identifiers $\mathcal{T}_i$ related by subclass relation is a poset forming a taxonomy of classes. Similarly also predicates, now in the role of types, are ordered in a poset. The meaning of class identifiers is established by their interpretations. The interpretatation of a class identifier c comprises the instances of a given class c es well as the instances of all c's sub-classes.

The individual triples include solely the individual identfifiers from $\mathcal{V}_i$ in places of S and O, and predicates $\mathcal{V}_p$ in the place of P component of a triple. The types of individual triples are product types $S * p * O$ where $S, O \in \mathcal{T}_i$ and $p \in \mathcal{T}_p$. The product types are in our data model of a KG written as a triple (S, p, O). The triple types are ordered by a subtype relation to form a poset. The subtype relation among the triple types is defined on the basis of the subtype relation among the classes and predicates. A more detailed formal definition of a KG is given in [28].

2.3 Typing rule language

We do not use standard typing rule language [22, 10] that includes a context $\varGamma$ where the types of the variables are stored. We use a meta-language that is rooted in first-order logic (abbr., FOL) but similar to the one used by Pierce in [22], in particular, for the representation of records and subtyping, as well as for the definition of the universal quantification.

The rules are composed of a set of premises and a conclusion. The premises can be expressions stating the set membership of an object, typing and subtyping judgements, or expressions in the FOL including the previous two forms. The expressions of FOL can express complex premises such as, for example, the conditions for the LUB types. The conclusion part of the rule is a typing or subtyping judgment.

The symbols used in a rule are grounded by stating their membership in the sets of identifiers ($\mathcal{V}$, $\mathcal{V}_i$ and $\mathcal{V}_t$) and triples ($\mathcal{T}$, $\mathcal{T}_i$ and $\mathcal{T}_t$) defined in Section 2.1. When we write $O \in S$ then we mean the *existence* of O in a set S. We use universal quantification $\forall O \in S,\ p(O)$ when we state that some property $p(O)$ holds for all objects O from S. The premises of the rule are treated from the left to the right. The quantification of the symbols binds the symbols up to the last premise unless defined differently by the parentheses.

In this section we present the basic typing of individual (ground) identifiers $\mathcal{V}_i$. Typing of literals $\mathcal{V}_l$ is described in Section 3.1. Nest, the rules for typing ground identifiers are presented in Section **??**. Section 3.3 further extends ground typing and subtyping to the complete poset set of individual identifiers $\mathcal{T}_i$. Finally, The sub-typing relation $\preceq$ is defined and the types of individual identifiers are presented. The rules for typing individual identifiers are further used for typing individual triples $\mathcal{V}_t$ in Section 5.

3 Typing identifiers

In this section, we present the basic typing of an individual (ground) identifiers $\mathcal{V}_i$. Typing of literals $\mathcal{V}_l$ is described in Section 3.1. Next, the rules for typing ground identifiers are presented in Section 3.2. Section 3.3 further extends ground typing and subtyping to the complete poset set of individual identifiers $\mathcal{T}_i$. The sub-typing relation $\preceq$ is defined, and typing rules for the individual identifiers are presented. The rules for typing individual identifiers are further used for typing individual triples $\mathcal{V}_t$ in Section 5.

3.1 Typing literals

Literal values $\mathcal{V}_l \subseteq \mathcal{V}_i$ are the instances of literal types $\mathcal{T}_l \subseteq \mathcal{T}_i$. The literal types $\mathcal{T}_l$ are provided by the RDF-Schema dictionary [25]. RDF-Schema defines a rich set of literal types, such as xsd:integer, xsd:string, or xsd:boolean.

The literals are composed of literal values and literal types. For example, the literal "365"^^xsd:integer includes the literal value "365" and a type xsd:integer. Typing of literals is defined by the following rule.

$$\frac{L \in \mathcal{V}_l \quad T \in \mathcal{T}_l \quad \text{"L"\^{}\^{}T} \in \Delta}{L : T} \tag{1}$$

The rule states that a literal value L is of a type T if a literal "L"^^T is an element of a KG Δ. A literal type T is referencing a type from the RDF-Schema dictionary.

3.2 Ground typing and subtyping of identifiers

The typing expression $V :_\downarrow T$ is a *ground typing* relation $:_\downarrow$ that links a value $V \in \mathcal{V}$ to a ground type $T \in \mathcal{T}$. The ground typing relation is a one-step typing relation based on typing stored in a KG.

In this section, we deal with the identifiers $I \in \mathcal{V}_i$, which are the values of type $T \in \mathcal{T}_i$. The ground types are directly linked to the ground identifiers via a stored typing relation in the form $(I, \text{rdf:type}, T) \in \Delta$. The expression $I :_\downarrow T$ states that a class identifier T is a ground type of an individual identifier I.

The subtyping expression $T_1 \preceq_\downarrow T_2$ defines a subtype relationship between the types T_1 and T_2. In the case we deal with the identifiers, the relation $T_1 \preceq_\downarrow T_2$ denotes the subclass relations since $T_1, T_2 \in \mathcal{T}_i$. The relation $\preceq_\downarrow$ is a one-step relation that is stored in Δ. Note that a unique notation for ground typing allows us to address differently the *stored* and the *derived* types of a KG.

The rule for the one-step typing relation $:_\downarrow$ is defined using the predicate rdf:type.

$$\frac{I \in \mathcal{V}_i \quad T \in \mathcal{T}_i \quad (I, \text{rdf:type}, T) \in \Delta}{I :_\downarrow T} \tag{2}$$

The individual entity I can have more than one stored type. By using Rule 2 as a generator, it synthesizes all types $T_j^{j\in[1,n]}$ such that $I :_\downarrow T_j^{j\in[1,n]}$.

If we want to obtain all valid ground types of I, the rule can be used either in some other rule that employs it as a generator, or we can update the above rule to generate a $\wedge$-type, including all the types of I as presented in Section 4.

The one-step subtyping relationship $\preceq_\downarrow$ is defined on classes by using the RDF predicate rdfs:subClassOf as follows.

$$\frac{C_1, C_2 \in \mathcal{T}_i \quad (C_1, \text{rdfs:subClassOf}, C_2) \in \Delta}{C_1 \preceq_\downarrow C_2} \tag{3}$$

The rule for the definition of the one-step subtyping relationship $\preceq_\downarrow$ is based on the predicate rdfs:subPropertyOf.

$$\frac{P_1, P_2 \in \mathcal{T}_p \quad (P_1, \text{rdfs:subPropertyOf}, P_2) \in \Delta}{P_1 \preceq_\downarrow P_2} \tag{4}$$

The predicates have in the above rule the role of types in the sense that they represent the names of the binary relations.

3.3 Typing and subtyping identifiers

The one-step relationship $\preceq_\downarrow$ is extended with the reflectivity, transitivity, and antisymmetry to obtain the subtyping relationship $\preceq$. The relation $\preceq$ forms a partial ordering of individual identifiers.

The ground typing relation $:_\downarrow$ is then extended with the *rule of subsumption* presented as Rule 11 to obtain a typing relation $:$. The rules are defined for the set of types $\mathcal{T}$ that includes the types of individual identifiers $\mathcal{T}_i$ and triple types $\mathcal{T}_t$.

First, the one-step relationship $\preceq_{\downarrow}$ is generalized to the relationship $\preceq$ defined over the type identifiers $\mathcal{T}_i$.

$$\frac{I_1, I_2 \in \mathcal{T}_i \quad I_1 \preceq_{\downarrow} I_2}{I_1 \preceq I_2} \tag{5}$$

Next, the subtyping relationship $\preceq$ is reflexive.

$$\frac{I \in \mathcal{T}_i}{I \preceq I} \tag{6}$$

The subtype relationship is also transitive.

$$\frac{I_1, I_2, I_3 \in \mathcal{T}_i \quad I_1 \preceq I_2 \quad I_2 \preceq I_3}{I_1 \preceq I_3} \tag{7}$$

Finally, the subtype relationship is antisymmetric, which is expressed using the following rule.

$$\frac{I_1, I_2 \in \mathcal{T}_i \quad I_1 \preceq I_2 \quad I_2 \preceq I_1}{I_1 = I_2} \tag{8}$$

As a consequence of the rules 6-8 the relation $\preceq$ defines a poset $(\mathcal{T}_i, \preceq)$.

Knowledge graphs include a special class $\top$ that represents the root class of the ontology. In RDF ontologies, $\top$ is usually represented by the predicate owl:Thing [11]. The following rule specifies that all class identifiers are more specific than $\top$.

$$\frac{\forall S \in \mathcal{T}_i}{S \preceq \top} \tag{9}$$

The one-step typing relation $:_{\downarrow}$ is now extended to the typing relation $:$ that takes into account the subtyping relation $\preceq$. The following rule states that a stored type is a type.

$$\frac{I \in \mathcal{V}_i \quad C \in \mathcal{T}_i \quad I :_{\downarrow} C}{I : C} \tag{10}$$

The link between the typing relation and the subtype relation is provided by adding a typing rule called *rule of subsumption* [22].

$$\frac{I \in \mathcal{T}_i \quad S, T \in \mathcal{T}_i \quad I : S \quad S \preceq T}{I : T} \tag{11}$$

4 Intersection and union types

The rules for the $\wedge$ and $\vee$-types presented in this section are general—they apply for the identifier types $\mathcal{T}_i$ and triple types $\mathcal{T}_t$.

The meaning of the $\wedge$ and $\vee$-types can be seen through their interpretations. The instances of the intersection type $T_1 \wedge T_2$ are objects belonging to both T_1 and T_2. The type $T_1 \wedge T_2$ is the greatest lower bound of the types T_1 and T_2. In general, $\wedge_{i=1}^{n} T_i$ is the greatest lower bound (abbr. GLB) of types $T_i^{\ i=1..n}$ [20, 21]. The instances of the type $\wedge_{i=1}^{n} T_i$ form a maximal set of objects that belong to all types T_i.

The union type is dual to the intersection type. The instances of the union type $T_1 \vee T_2$ are objects from the interpretations of both types, T_1 and T_2. The type $T_1 \vee T_2$ denotes the least upper bound of types T_1 and T_2. A general form of union type is $\vee_{i=1}^n T_i$. The interpretation of $\vee_{i=1}^n T_i$ represent the minimal set of instances that include the instances of all types $T_i^{\ i=1..n}$.

The interpretation of a type $\wedge_{i=1}^n T_i$ is included in interpretation of every particular type T_i which means $\wedge_{i=1}^n T_i \preceq T_i$. This is stated by the following rule.

$$\frac{T_i^{\ i=1..n} \in \mathcal{T}}{\wedge_{i=1}^n T_i \preceq T_i^{\ i=1..n}} \tag{12}$$

Further, the following rule states that if the type S is a subtype of every type $T_i^{\ i=1..n}$ then S is a subtype of $\wedge_{i=1}^n T_i$.

$$\frac{S \in \mathcal{T} \quad T_i^{\ i=1..n} \in \mathcal{T} \quad S \preceq T_i^{\ i=1..n}}{S \preceq \wedge_{i=1}^n T_i} \tag{13}$$

The opposite to the above rule, the following rule states the necessary conditions that must be met so that a type T is a supertype of a type $\wedge_{i=1}^n T_i$. It is enough that there is one type $S_i \preceq T$ for the intersection of $S_i^{\ i=1..n}$ to be included in T.

$$\frac{T \in \mathcal{T} \quad S_i^{\ i=1..n} \in \mathcal{T} \quad S \in \{S_i\}_{i=1}^n \quad S \preceq T}{\wedge_{i=1}^n S_i \preceq T} \tag{14}$$

The duality of the intersection and union types can also be seen from the duality of the rules for the $\wedge$ and $\vee$-types.

The union type $\vee_{i=1}^n T_i$ is the least upper bound of types $T_i^{\ i=1..n}$ [20]. This means that all types $T_i^{\ i=1..n}$ are subtypes of their union.

$$\frac{T_i^{\ i=1..n} \in \mathcal{T}}{T_i^{\ i=1..n} \preceq \vee_{i=1}^n T_i} \tag{15}$$

The following rule defines the necessary conditions to be met for a type T to be a supertype of a type $\vee_{i=1}^n S_i$.

$$\frac{T \in \mathcal{T} \quad S_i^{\ i=1..n} \in \mathcal{T} \quad S_i^{\ i=1..n} \preceq T}{\vee_{i=1}^n S_i \preceq T} \tag{16}$$

Again, the opposite rule defines the premises that must hold so that T is a subtype of a type $\vee_{i=1}^n S_i$.

$$\frac{T \in \mathcal{T} \quad S_i^{\ i=1..n} \in \mathcal{T} \quad S \in \{S_i\}_{i=1}^n \quad T \preceq S}{T \preceq \vee_{i=1}^n S_i} \tag{17}$$

Note that besides checking the subtype relation between a type treated as a whole and some logical type, Rules 12-17 can be used to check the subtyping among arbitrary logical types.

4.1 The join and meet types

The $\vee$ and $\wedge$ types are logical types defined on the basis of their interpretations, i.e., the sets of instances. Given two types T and S we have a least upper bound $S \vee T$, and a greatest lower bound $S \wedge T$ types where $S \vee T$ denotes a minimal set of objects that are of type S *or* T (or both), and $S \wedge T$ denotes a maximal set of objects that are of type S *and* T.

A KG includes a stored poset of classes and triple types that represent types of the individual identifiers and ground triples. The poset can be used to compute a join $S \sqcup T$ and a meet $S \sqcap T$ of the parameter types. Usual definition of the join and meet operators is by using a least upper bound and a greatest lower bound if they exist [22], respectively. However, in a KG we are also interested in the upper bound and lower bound types [6]. Let us present an example.

Example 1. Let $P = (U, \preceq)$ be a partialy oredered set P such that $U = \{a, b, c, d, e\}$ and the relation $\preceq = \{a \preceq c, a \preceq d, b \preceq c, b \preceq d, c \preceq e, d \preceq e\}$. The upper bounds of $S = \{a, b\}$ are the elements c and d. Since there is no lower upper bounds, the upper bounds $\{c, d\}$ are minimal upper bounds. The least upper bound of S is e.

In the case that we remove the element e from P then P does not have a least upper bound but it still has two minimal upper bounds c and d. $\square$

The least upper bound (abbr., LUB) is by definition one element. It has to be related to all upper bounds via the relationship $\preceq$. On the other hand, the most interesting upper and lower bounds are minimal upper bounds (abbr., MUB) and maximal lower bounds (abbr. MLB) [16]. They can be lower than the least upper bound and higher than the greatest lower bound, respectively. They represent more detailed information about the parameter set of types S than the LUB type of S.

The join $J = \sqcup_{i=1}^{n} T_i$ is a set of MUB types $J_j^{j \in [1,m]} \in J$ such that J_j is an upper bound with $T_i{}^{i=1..n} \preceq J_j$, and there is no such L where $T_i{}^{i=1..n} \preceq L$ without also having $J_j \preceq L$. Since we have a top type $\top$ defined in a KG, the join of arbitrary two types always exists.

The meet of types $T_i{}^{i=1..n}]$, $M = \sqcap_{i=1}^{n} T_i]$, is a set of the maximal lower bound types $M_j^{j \in [1,m]} \in M$ such that M_j is lower bound with $M_j \preceq T_i{}^{i=1..n}$, and all other lower bounds U with $U \preceq T_i{}^{i=1..n}$ entail $U \preceq M_j$. Note that the meet of the set of types from a KG does not always exist.

The join type is related to the $\vee$-type. Given a set of types $\{T_i\}_{i=1}^{n}$, the join $J = \sqcup_{i=1}^{n} T_i$ is a set of types $J_j^{j \in [1,m]} \in J$ that are the minimal upper bounds such that $T_i{}^{i=1..n} \preceq J_j$. On the other hand, Rule 15 for the $\vee$-types states $T_i{}^{i=1..n} \preceq \vee_{i=1}^{n} T_i]$. However, the join type and $\vee$-type differ in the interpretation.

$$[\![\vee_{i=1}^{n} T_i]\!]_{\Delta} = \bigcup_{i=1}^{n} [\![T_i]\!]_{\Delta} \subseteq \bigcup_{j=1}^{m} [\![J_j]\!]_{\Delta} = [\![\sqcup_{i=1}^{n} T_i]\!]_{\Delta}$$

While the interpretation of the type $\vee_{i=1}^{n} T_i]$ includes precisely the instances of all T_i, the interpretation of the type $\sqcup_{i=1}^{n} T_i$ contains the instances of minimal upper bound types. The interpretation of $\sqcup_{i=1}^{n} T_i$ can include interpretations of classes that are not among $T_i{}^{i=1..n}$.

A meet type of $T_i\ ^{i=1..n}$ may not exist in a poset of types from a KG. In general, the meet types $M = \sqcap_{i=1}^{n} T_i$ exist in a class ontology if the types $T_i\ ^{i=1..n}$ are *bounded below* [22] which means that there exists a type L such that $L \preceq T_i$ for all i. The meet types are not frequent on the lower levels of a class ontology from a KG.

As in the case of the $\vee$-type and the join type, the semantics of the $\wedge$-type is similar to the semantics of the meet type. An $\wedge$-type is a type that implements logical view of the greatest lower bound type. Differently, the meet types are based on the concrete poset of KG types and represent concrete types though their interpretation is contained in the interpretation of a $\wedge$-type. The type $\wedge_{i=1}^{n} T_i$ denotes the intersection $\bigcap [\![T_i]\!]_{\Delta}$ while the interpretation of a meet types $M_j\ ^{j=1..m} \in \sqcap_{i=1}^{n} T_i$ includes the union $\cup_{j\in[1,m]} [\![M_j]\!]_{\Delta}$. Since $M_j \preceq T_i\ ^{i=1..n}$, then $[\![M_j]\!]_{\Delta} \subseteq [\![T_i\ ^{i=1..n}]\!]_{\Delta}$. Hence, $[\![\sqcap[T_i\ ^{i=1..n}]]\!]_{\Delta} = \cup_{j=1}^{m} [\![M_j]\!]_{\Delta}$

Note that the instances of the meet types are in the intersection of the instances of types $T_i\ ^{i=1..n}$. The set $\bigcap [\![T_i]\!]_{\Delta}$ can also include objects that are not instances of any meet type from M. Hence,

$$[\![\wedge_{i=1}^{n} T_i]\!]_{\Delta} = \bigcap_{i=1}^{n} [\![T_i]\!]_{\Delta} \supseteq \bigcup_{j=1}^{m} [\![M_j]\!]_{\Delta} = [\![\sqcap_{i=1}^{n} T_i]\!]_{\Delta}.$$

In type-checking the ground triples, the join types are used in the procedure for checking the types derived bottom-up against the stored schema of a KG as presented in Section 5.1. The join as well as meet types are useful in the procedure for type-checking basic graph patterns [27]. The $\vee$ and $\wedge$-types are logical types that can be simplified in the typing positions of a graph pattern by using typing rules, and can be approximated by using join and meet types to obtain a more precise concrete type of a graph pattern variable.

4.2 Typing identifiers with $\wedge$ and $\vee$-types

The $\wedge$ and $\vee$-types can model the available choices in selection of the domain and range types of a triple type. The choices depend on the selected model (e.g., RDF-Schema). As usual for the expressions including a variant of $\cup$ and $\cap$ operators more complex expressions can be transformed by moving $\cup$ and $\cap$ either inside expression or towards the outside of an expression. We define the rules for these transformations only if they are needed for typing ground triples.

We start with a rule for gathering the ground types of a ground identifier $I \in \mathcal{V}_i$. The ground types are gathered in the premise of the rule by selecting individual ground types as presented in Section 3.2.

$$\frac{I \in \mathcal{V}_i \quad T_i\ ^{i=1..n} \in \mathcal{T}_i \quad I :_{\downarrow} T_i\ ^{i=1..n}}{I :_{\downarrow} \wedge_{i=1}^{n} T_i} \tag{18}$$

The ground type $\wedge_{i=1}^{n} T_i$ can include pairs of types $T_i \preceq T_k$ with $i \neq k$. Depending on the further use, we can either compute the minimal or the maximal elements from the poset $\{T_i\}_{i=1}^{n}$ with respect to $\preceq$ [6]. The supertypes of the minimal elements of $\{T_i\}_{i=1}^{n}$ include all valid types of I. Hence, we use the set of minimal elements from

$\{T_i\}_{i=1}^n$ as the starting point to explore the relations between the ground types and the schema types of a triple t.

Given a poset of types $(\{T_i\}_{i=1}^n, \preceq)$ the operator MIN retains types $S_j^{\ j=1..m} \in \{T_i\}_{i=1}^n$ such that $\nexists T_i^{\ i=1..n}(T_k \prec S_j)$. All pairs of types $S_k, S_l \in \{S_j\}_{j=1}^m$ with $k \neq l$ are *incomparable*, i.e., $S_k \not\sim S_l \equiv S_k \not\preceq S_l \wedge S_k \not\succeq S_l$. The logical rule for deriving minimal type from $\{T_i\}_{i=1}^n$ is as follows.

$$\frac{I \in \mathcal{V}_i \quad I :_{\downarrow} \wedge_{i=1}^n T_i}{I :_{\Downarrow} \wedge\{S \mid S \in \{T_i\}_{i=1}^n \quad \forall i \in [1,n],\ S \preceq T_i \vee S \not\sim T_i\}} \tag{19}$$

The rule says that the types S gathered with the $\wedge$-type are minimal types of a ground type $\wedge_{i=1}^n T_i$. A particular type S is minimal since all other T_i types are either more general or equal ($\succeq$), or not related to S.

Let us now present the typing rules that, given $I \in \mathcal{V}_i$, determine the join of types from $\{T_i\}_{i=1}^n$ as $I : \sqcup_{i=1}^n T_i$. Recall from Section 4.1 that we defined the operation join as the minimal upper bound of a set $\{T_i\}_{i=1}^n$. A join $\sqcup_{i=1}^n T_i$ is a set of minimal upper bounds $\{S_j\}_{j=1}^m$ that are related to all types $T_i^{\ /i \in [1,n]}$ via $\preceq$, and are minimal.

$$\frac{I \in \mathcal{V}_i,\ I :_{\downarrow} \wedge_{i=1}^n T_i}{I :_{\sqcup} \wedge\{S \mid S \in \mathcal{T}_i,\ T_i^{\ i=1..n} \preceq S \wedge \forall P \in \mathcal{T}_i,\ (T_i^{\ i=1..n} \preceq P \wedge S \preceq P) \vee P \not\sim S\}} \tag{20}$$

The MUB types are used as the starting points for searching a correct schema type. In case the predicate has two different definitions in two different contextes, then the paths from a MUB type to the schema of a KG determine the correct schema type. The details are presented in Section 5.2.

5 Typing triples

A type of a triple $(s, p, o) \in \mathcal{V}_t$ is a is a product type $D * p * R$ where $s : D$ and $o : R$ holds. In our model, the triple types are represented by a triple $(D, p, R) \in \mathcal{T}_t$. The types D and R are type expressions that represent either a class identifier or an expression composed of classes related by $\wedge$ and/or $\vee$ operators. The available logical operators depend on the schema languege of a KG. For now, we assume the schema language is RDF-Schema.

The $\wedge$-types reflect the semantics of RDF-Schema [25] which permits the definition of multiple domains and ranges of the predicate p that are linked with $\wedge$-types. Consequently, each predicate p has exactly one triple type of the form

$$(\wedge_{i=1}^n D_i^{\ i=1..n}, p, \wedge_{j=1}^m R_j).$$

Example 2. If p has two domains D_1 and D_2, and a single range type R then the type corresponding to p is $(D_1 \wedge D_2, p, R)$. □

The interpretation of a triple type (D, p, R) is defined as follows.

$$[\![(D, p, R)]\!]_\Delta = \{(s, p, o) \mid s \in [\![D]\!]_\Delta \wedge o \in [\![R]\!]_\Delta\}$$

The subtype relationship among the triple types is defined on the basis of the subtype relationship among classes, $\wedge$ and $\vee$-types, and predicates. The following rule defines the relationship $\preceq$ beween two triple types.

$$\frac{T_1 \in \mathcal{T}_t,\ T_1 = (D_1, p_1, R_1) \quad T_2 \in \mathcal{T}_t,\ T_2 = (D_2, p_2, R_2) \quad D_1 \preceq D_2 \quad p_1 \preceq p_2 \quad R_1 \preceq R_2}{T_1 \preceq T_2} \tag{21}$$

The above Rule 21 handles the $\wedge$-types of the subject and object through Rules 12-14.

5.1 Ground types of a triple

The ground types of a triple t are either a stored ground type, a minimal ground type, or a join ground type. The stored ground type includes types that are stored in a KG. The minimal ground type then consists solely of the minimal types from the stored ground types. Finally, the join ground type is the conjunction of minimal upper bound types [16].

A ground type of an individual identifier I is a class C related to I by one-step type relationship $:_\downarrow$, as presented by Rule 2. In terms of the concepts of a knowledge graph, C and I are related by the relationship rdf:type.

A ground type of a triple $t = (I_s, p, I_o)$ is a product type $T * p * S$ that we represent as a triple (T, p, S). A triple type includes the ground types of t's components I_s and I_o, and the property p, which now has the role of a type. A ground type of a triple is defined by the following rule.

$$\frac{t \in \mathcal{V}_t,\ t = (I_s, p, I_o) \quad T, S \in \mathcal{T}_i \quad I_s :_\downarrow T \quad I_o :_\downarrow S \quad p :_\downarrow \text{rdf:Property}}{t :_\downarrow (T, p, S)} \tag{22}$$

The type T is either a class identifier or a $\wedge$-type composed of a conjunction of class identifiers. The predicates are treated differently from the subject and object components of triples. The predicates have the role of types while they are instances of rdf:Property.

The minimal ground type of a triple t can be obtained by using the minimal ground types of the triple components.

$$\frac{t \in \mathcal{V}_t,\ t = (I_s, p, I_o) \quad T, S \in \mathcal{T}_i \quad I_s :_\Downarrow T \quad I_o :_\Downarrow S \quad p :_\downarrow \text{rdf:Property}}{t :_\downarrow (T, p, S)} \tag{23}$$

The component types T and S of the minimal ground type (T, p, S) can represent $\wedge$-types. The following rule transforms a triple type including $\wedge$-types into a $\wedge$-type of simple triple types composed of class identifiers in place of S and O components. The rule is expressed in a general form by using the typing relation ":", which can be replaced by any labeled typing relation.

$$\frac{t \in \mathcal{V}_t \quad t : (\wedge_{i=1}^{n} S_i, p, \wedge_{j=1}^{m} R_j)}{t : \bigwedge_{i=1..n, j=1..m} (S_i, p, R_j)} \tag{24}$$

The type in the conclusion of the rule is constructed by the Cartesian product of the sets of types belonging to types of S and O components. Since each of the types S_i and R_j is valid for the components S and O, respectively, then also the triple types from the conclusion of the rule are valid.

The above decomposition of a triple type into a set of triple types is useful when we check the ground types against the schema triple types to select the valid schema triple type of a triple. This is detailed in Section 5.2.

Finally, the join of a set of ground types is a set of minimal upper bound types. Similarly to the previous two rules, the join is defined on the basis of joins of triple type components S and O.

$$\frac{t \in \mathcal{V}_t,\ t = (I_s, p, I_o) \quad T, S \in \mathcal{T}_i \quad I_s :_{\sqcup} T \quad I_o :_{\sqcup} S \quad p :_{\downarrow} \text{rdf:Property}}{t :_{\downarrow} (T, p, S)} \tag{25}$$

The join type (T, p, S) includes in the components S and O the $\wedge$-types comprising one or multiple MUB classes. When we convert this type into a conjunction of single MUB triple types, then each MUB triple type stands for all ground triple types of t.

We can easily see that each particular triple type is an MUB type since it includes MUB types in its components. Since the MUB types of the components are incomparable by $\preceq$ then also the MUB triple types obtained by Rule 25 are incomparable.

All rules defined for the ground triple types rely on inferring the types of their components. The reason for this are the stored typing of individual identifiers as well as the stored poset relation $\preceq$, which are solely defined on classes.

5.2 Schema triple types

The schema types are types of triples defined by a variant of KG schema. The schema definition language currently used in KGs is either RDF-Schema [25] or RDF-Schema combined with OWL [9] vocabulary. In this section, we present the semantics of both approaches.

We do not expect that the domain and range of the predicate are defined for each particular predicate p. They can be inherited from the super-predicates of p. Hence, a predicate p has the domain and range defined either directly, when domain and range are defined for the predicate p, or indirectly, if the domain and range are defined for p's super-predicates and inherited by the predicate p.

The rules for the derivation of the schema triple type of a given triple t are presented for two different schema definition languages. In Section 5.2.1 we present the derivation of schema types when the RDF-Schema is used. Further, in Section 5.2.2 we define the typing rules for KGs that use RDF-Schema together with $\wedge$ and $\vee$ types.

5.2.1 KGs with RDF-Schema. When a KG is restricted by using RDF-Schema, we can specify one or more domain and range types. The semantics of RDF-Schema [26] interprets multiple domains and ranges with $\wedge$-type. If p has two domains T_1 and T_2 then p can link subjects I that are of type T_1 *and* T_2. The domain type of p is then $T_1 \wedge T_2$, or, in terms of OWL [19], owl:intersectionOf(T_1 T_2). The RDF-Schema does not allow the definition of the domain of a predicate with the $\vee$-type. Consequently, each predicate can have only one meaning.

We first determine *all* valid schema types for a given triple $t = (s, p, o)$. A schema triple type comprises a predicate p and the domain and range types linked to predicates $p' \succeq p$. The domain and range types can be defined for the predicate p and/or inherited from the predicates $p' \succ p$. In addition, the domain and range types can be, in general, inherited from two different super-predicates of p.

$$\frac{\begin{array}{c} t \in \mathcal{V}_i,\, t = (s, p, o) \quad p_1, p_2 \in \mathcal{T}_p,\, p \preceq p_1 \preceq p_2 \\ T_i{}^{i=1..n} \in \mathcal{T},\, (p_1, \text{domain}, T_i) \in \Delta \quad S_j{}^{j=1..m} \in \mathcal{T},\, (p_2, \text{range}, S_j) \in \Delta \end{array}}{t :_{\uparrow} (\bigwedge_{i=1}^{n} T_i, p, \bigwedge_{j=1}^{m} S_j)} \tag{26}$$

The above rule generates pairs of types of the domain and range of a predicate p. We allow that a domain and range types are defined for different predicates $p_1, p_2 \preceq p$ since such a situation can appear in a KG. However, we have to be careful that the rules of inheritance are respected. We can only inherit from p_1 and p_2 that are related by subtype relation: $p \preceq p_1 \preceq p_2$. This condition restricts the domain and range to be defined on the same path from p to some maximal element m of the predicate p poset.

The above Rule 26 generates all valid schema types of a triple t. From the set of all valid stored types of t we select the subset including only the minimal types. The following rule is a logical judgment for a minimal schema type of a t.

$$\frac{t \in \mathcal{T}_i \quad T \in \mathcal{T},\, t :_{\uparrow} T \quad S_i{}^{i \in [1,n]} \in \mathcal{T},\, t :_{\uparrow} S_i \quad T \preceq S_i \vee T \not\sim S_i}{t :_{\uparrow} T} \tag{27}$$

The first part of the premise says that t is a ground triple and there exists $T \in \mathcal{T}$ which is a type of t. The second part of the premise requires that T is the minimal type of all types S of t. In other words, there is no type $S_i{}^{i=1..n}$ of t that is a subtype of T. Hence, T is the minimal type of the stored triple types of t.

Note that if the schema is defined by using RDF-Schema, and the stored schema typing is correct, then the condition $T \sim S_i$ is always $true$, and Rule 27 generates exactly one minimal type. Furthermore, under the restrictions of RDF-Schema, we can not define a predicate p with two meanings. If we were to specify two different domains or ranges of p, then the reasoner would treat the domain and range types as $\wedge$-types. Each instance of the domain (range) type has to be an instance of all specified types of the domain (range).

5.2.2 KGs with the contextual representation. The collective findings of the research in the area of Cognitive Science [12] shows that natural language is inherently contextual, and the context is essential in the human representation of knowledge and reasoning.

While the current trend is to enforce exactly one meaning of a predicate in KGs, the contextual representation and reasoning allow the definition of multiple senses of a predicate. There are many motivations for adopting contextual representation and reasoning in a KG. First, with the evolution of KGs, there are many examples where a

KG is represented in a modular way, splitting the dataset into parts that correspond to the contexts. The meaning of a predicate can be different in different contexts, while the reasoner is able to disambiguate among the different meanings of a predicate. The practical examples of KGs using contexts include the named graphs in DBpedia [1], Wikidata [30], and Yago [11]. Another example is Cyc [23] that uses microtheories to represent different contexts. Similar to Cyc, Scone [8] is a KR system that can define spaces (contexts) and uses contextual reasoning.

The second motivation for using contextual representation of KGs is the problem of a predicate with multiple senses represented with multiple sub-predicates. In a query—whether expressed in natural language, logic or as a database query—it is difficult to disambiguate the correct sub-predicate for the particular query. The alternative is that a user must explicitly select a correct sense of a predicate (i.e., a sub-predicate) in the query.

Finally, a predicate can be compared to a mathematical function since it represents a binary relation. In mathematics, a function is not represented by its name only, but with a function type including, besides the function name, also the types of its domain and range. Types of functions can disambiguate among the different functions with the same name but different domain and range types. Similarly, the types can be effectively used to disambiguate the meaning of the predicate in a KG.

To be able to study the behavior of KG predicates with multiple senses in the presence of triple types, we propose a minimal KR schema language that includes the triple types stored in a KG as triples. In case there is more than one triple type including a predicate p, then these are treated as alternatives. For example, if a KG includes triples (T_1, p, T_2) and (T_3, p, T_4), where $T_j^{\ j=1..4} \in \mathcal{T}_i$, then the type of ground triples including p is $T_1 * p * T_2 \vee T_3 * p * T_4$.

Further, the proposed KR language can use $\wedge$ and $\vee$-types in subject and object components of triple types. The $\wedge$ and $\vee$-types are often implemented in KGs in the form of OWL type constructors owl:intersectionOf and owl:unionOf[1] The use of $\vee$-type in place of the domain or range type is redundant since it can be expressed with multiple triple types. Hence, the *minimal KR schema language* includes solely the triple types of the form

$$\bigwedge_{i=1}^{n} D_i * p * \bigwedge_{j=1}^{m} R_j.$$

Let us now present the rules that derive the stored types of a ground triple t in the case our minimal KR schema language is used for the definition of the KG schema. First, the following Rule 28 generates all valid alternatives of stored triple types given a triple t. Note that D_i and R_i are the types of domains and ranges of p that can stand for a $\wedge$-type.

$$\frac{t \in \mathcal{V}_t,\ t = (s,p,o) \quad T_i^{\ i=1..n} \in \mathcal{T}_t,\ T_i = (D_i,p,R_i) \quad T_i^{\ i=1..n} \in \Delta}{t :_{\uparrow} \bigvee_{i=1}^{n} T_i} \tag{28}$$

[1] The OWL union and intersection type constructors are employed mostly in domain-specific KGs like biomedical and genomic ontologies. In these scientific fields, the knowledge base includes large ontologies where new classes can be defined as logical combinations of existing classes.

A schema triple type $\bigvee_{i=1}^{n} T_i$ of t is a disjunction of all valid schema triple types of t. Similar to Rule 19, which is defined to find minimal types of a set of classes, the following Rule 29 generates the disjunction of minimal schema types.

$$\frac{t \in \mathcal{V}_t \quad t :_{\uparrow} \bigvee_{i=1}^{n} T_i}{t :_{\Downarrow} \bigvee\{S \mid S \in \{T_i\}_{i=1}^{n} \wedge \forall i = 1..n,\, S \preceq T_i \vee S \not\sim T_i\}} \tag{29}$$

The rule says that the triple types S are minimal schema types of a schema type $\bigvee_{i=1}^{n} T_i$. S is minimal since all other types T_i are either more general or equal ($\succeq$), or not related to S.

5.3 Typing a triple

Before we present the final step in typing a triple t, an overview of the work done so far is given. First, we derive the ground type of the triple t and then infer the minimal upper bound of the ground type. The ground typing inspects all ground types of t's components. In case the MUB type is close to $\top$ type then there is an outlier in the set of ground types, it can be revealed in computation of the MUB type.

Second, the schema type of t is derived from the schema of a KG. In case we use RDF-Schema semantics of a KG then the rules infer a single minimal schema type. If there are more than one schema types than the domains and ranges from the different definitions are merged into one $\wedge$-type. At this point we can not verify if there are any errors in types from the domain and/or range of a predicate. in schema (stored) typing. In case we use RDF-Schema with $\wedge$ and $\vee$ types, then the rules can infer multiple disjunctive schema types. Also in this case there are no additional constraints that could be verified.

Let us now present final typing of t by relating the ground type and schema types (minimal schema triple types) via subtyping relation. If the RDF-Schema semantics is used then we have a single schema type which should be related to the ground type of t. The following Rule 30 derives the final type of t under RDF-Schema semantics.

$$\frac{t \in \mathcal{T}_i \quad T \in \mathcal{T}_t,\, t :_{\downarrow} T \quad S \in \mathcal{T}_t,\, t :_{\uparrow} S \quad T \preceq S}{t : S} \tag{30}$$

The type of t is computed by first deriving the ground type T and the schema type S of t. S is the final type of t if T is a subtype of S. In case $T \not\preceq S$ then the ground type $T = \wedge[T_i^{i \in [1,n}]$ includes at least one $T_i \not\preceq S$.

$$\frac{t \in \mathcal{T}_i \quad t :_{\downarrow} T \quad t :_{\uparrow} \bigvee_{j=1}^{m} S_j}{t : \bigvee\{S_j \mid j \in [1,m], T \preceq S_j\}} \tag{31}$$

...

6 Implementation

7 Related work

Most of the related work is cited in the text when presenting the particular topic. In this section, we present only the work that is not directly related to the presented work but represents a contribution to the typing of the knowledge graphs.

The entity typing and type inference deal with predicting and inferring the types (classes) of entities that are either missing or incorrect. The automatic type assignment can use logic-based assignment where a reasoner infers the type of an entity from the schema [14]. Alternatively, the rule-based inference can be employed on the rules defined as a schema by knowledge engineers [13]. Reasoners use them automatically. Another alternative is the use of ML-based type prediction [31]. Various techniques can be used, like entity embeddings and graph neural networks, to produce embeddings to be fed into the classifier.

The schema-based type checking is about the verification of RDF-Schema [29] and OWL [9] rules and constraints against the data (ABox) and schema (TBox) parts of a KG [2, 14, 18]. The domain and range types of predicates are checked to determine whether they are correctly interpreted among the ground triples of a KG. The consistency of subtyping and inheritance is verified in the KG schema and in the data. Similarly, the disjointness of types and other OWL constraints is checked. Most of the presented themes are covered by tools based on SHACL [15] and ShEx [3].

Type checking in query answering ensures that the variables and results of a query over a KG are consistent with the schema of a KG [33, 32]. A type-checker for queries requires that a query respects class hierarchies, domain/range constraints, and disjointness rules. The type information can also be used to improve the query optimization and execution. In [17], they propose to leverage type information to optimize query execution and filter semantically invalid results. In most cases, the presented approaches use fragments of types that are adapted for the particular problem.

8 Conclusions

Typing of a knowledge graph can serve many theoretical and practical purposes. First, typing of a KG can be extended to type-checking that can identify inconsistencies in a KG. Next, typing a KG can be used as the basis for typing basic graph patterns and SparQL, and with this, they also can be used in tasks such as the query optimization and reasoning.

...

References

1. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2007.

2. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *Description Logic Handbook*. Cambridge University Press, 2002.
3. I. Boneva, J. E. L. Gayo, E. G. Prud'hommeaux, and S. Staworko. Shape expressions schemas, 2015.
4. R. J. Brachman and J. G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9:171–216, 1985.
5. J. J. Carroll, C. Bizer, P. Hayes, and P. Stickler. Named graphs, provenance and trust. In *Proceedings of the 14th International Conference on World Wide Web (WWW '05)*, pages 613–622. ACM, 2005.
6. B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 2nd edition, 2002.
7. J. Dunfield and N. Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021.
8. S. E. Fahlman. Using scone's multiple-context mechanism to emulate human-like reasoning. In *Advances in Cognitive Systems, Papers from the 2011 AAAI Fall Symposium, Arlington, Virginia, USA, November 4-6, 2011*, volume FS-11-01 of *AAAI Technical Report*. AAAI, 2011.
9. O. W. Group. Owl 2 web ontology language. W3C Recommendation, Dec. 2012.
10. J. R. Hindley. *Basic simple type theory*. Cambridge University Press, USA, 1997.
11. J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194(0):28 – 61, 2013. Artificial Intelligence, Wikipedia and Semi-Structured Resources.
12. D. L. Hollister, A. J. Gonzalez, and J. Hollister. Contextual reasoning in human cognition and its implications for artificial intelligence systems. In *Modeling and Using Context (CONTEXT 2017)*, volume 10257 of *Lecture Notes in Computer Science*, pages 599–608. Springer, 2017.
13. I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. Swrl: A semantic web rule language combining OWL and ruleml. In *W3C Member Submission*, 2004.
14. I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26, 2003.
15. H. Knublauch and D. Kontokostas. Shapes constraint language (SHACL). World Wide Web Consortium (W3C) Recommendation, July 2017. Accessed: 2024-09-07.
16. S. B. Knudstorp. Tho modal logic of minimal upper bounds, 2024.
17. I. Kollia and B. Glimm. Optimizing sparql query answering over owl ontologies. *Journal of Artificial Intelligence Research*, 48:253–303, 2013.
18. B. Motik, P. F. Patel-Schneider, and B. Parsia. OWL 2 web ontology language: Structural specification and functional-style syntax. W3C Recommendation, Dec. 2012.
19. Owl 2 web ontology language. http://www.w3.org/TR/owl2-overview/, 2012.
20. B. C. Pierce. Programming with intersection types, union types, and polymorphism, 1991.
21. B. C. Pierce. Intersection types and bounded polymorphism, 1996.
22. B. C. Pierce. *Types and Programming Languages*. MIT Press, 1 edition, Feb. 2002.
23. D. Ramachandran, P. Reagan, and K. Goolsbey. First-orderized researchcyc: Expressivity and efficiency in a common-sense ontology. In *AAAI Reports*. AAAI, 2005.
24. Resource description framework (rdf). http://www.w3.org/RDF/, 2004.
25. Rdf schema. http://www.w3.org/TR/rdf-schema/, 2004.
26. Rdf schema. https://www.w3.org/TR/rdf-mt/, 2004.
27. I. Savnik. Type-checking graph patterns. Technical Report Technical Report (In preparation), FAMNIT, University of Primorska, 2025.
28. I. Savnik, K. Nitta, R. Skrekovski, and N. Augsten. Type-based computation of knowledge graph statistics. *Annals of Mathematics and Artificial Intelligence*, 2025.

29. D. Tomaszuk and T. Haudebourg. Rdf 1.2 schema. W3C Working Draft, Apr. 2024.
30. D. Vrandečić and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.
31. Y. Yaghoobzadeh, H. Adel, and H. Schütze. Corpus-level fine-grained entity typing. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, pages 940–949, 2018.
32. L. Zhang, Y. Zhao, and L. Zhang. Entity type saturation: A technique for type checking in sparql query answering. *Semantic Web*, 10(1):1–19, 2019.
33. Y. Zhao, H. Wang, L. Zhang, and L. Zhang. Type checking and testing of sparql queries. In *Proceedings of the 16th International Semantic Web Conference (ISWC 2017)*, pages 526–541. Springer, 2017.