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 Δ. We use Δ 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 Γ 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] for the representation of records and subtyping.

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 the requirements for the LUB and GLB triple 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.

Similar to [7], we differ between two interpretations of rules. First, the *generator* view of rules is the forward interpretation where rules infer the types from the types derived by premises. Second, the *type-checking* view of the rules is the backward interpretation. Given the expression and its type, the rules are applied backwards to verify that the expression has that type.

3 Typing identifiers

In this section we present the basic typing of ground identifiers $\mathcal{V}_i$. The rules for typing ground identifiers are further used for typing ground triples $\mathcal{V}_t$ in Section 5. Typing of literals $\mathcal{V}_l$ is described in Section 3.1. The rules for deriving the stored types of ground identifiers are given in Section 3.2. Finally, the sub-typing relation $\preceq$ is defined for the class identifiers $\mathcal{T}_i$ and the types of ground identifiers are presented in Section 3.3.

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 $\mathcal{V}_l$ are provided by the RDF-Schema dictionary [25]. RDF-Schema defines a list 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 gound 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 types. 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 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 class identifiers. The ground typing relation $:_{\downarrow}$ is then extended with the *rule of subsumption* presented as Rule 11 to obtain a typing relation :.

First, the one-step relationship $\preceq_{\downarrow}$ is generalized to the relationship $\preceq$ defined over class 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.

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

The subtype relationship is also transitive.

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

antisymmetric 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 $\preceq$ is a poset.

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 specifys 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 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

– CHECK: upper and lower bounds in the following rules.

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, $\wedge[T_i^{i\in[1,n]}]$ is the greatest lower bound (abbr. GLB) of types $T_i^{i\in[1,n]}$ [20, 21]. The instances of the type $\wedge[T_i^{i\in[1,n]}]$ form a maximal set of objects that belong to all types T_i.

The rules for the $\wedge$ and $\vee$ types presented in this section are general—they apply for the identifier types $\mathcal{I}_c$ and triple types $\mathcal{T}_t$. The set of types $\mathcal{T} = \mathcal{I}_c \cup \mathcal{T}_t$ is used to ground the types in the rules.

The interpretation of a type $\wedge[T_i^{i\in[1,n]}]$ in included in interpretation of every particular type T_i. Hence, the type $\wedge[T_i^{i\in[1,n]}]$ is the greatest lower bound of types $T_i^{i\in[1,n]}$. This is stated by the following rule.

$$\frac{T_i^{i\in[1,n]} \in \mathcal{T}}{\wedge[T_i^{i\in[1,n]}] \preceq T_i^{i\in[1,n]}} \tag{12}$$

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

$$\frac{S \in \mathcal{T} \quad T_i^{i\in[1,n]} \in \mathcal{T} \quad S \preceq T_i^{i\in[1,n]}}{S \preceq \wedge[T_i^{i\in[1,n]}]} \tag{13}$$

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 $\wedge$-type $\wedge[S_i^{i\in[1,n]}]$.

$$\frac{T \in \mathcal{T} \quad S_i^{i\in[1,n]} \in \mathcal{T} \quad S \in \{S_i^{i\in[1,n]}\} \quad S \preceq T}{\wedge[S_i^{i\in[1,n]}] \preceq T} \tag{14}$$

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

The instances from the union type $T_1 \vee T_2$ are either the instances of T_1 or T_2, or the instances of both types. Therefore, $\vee[T_i^{i\in[1,n]}]$ is the least upper bound of types $T_i^{i\in[1,n]}$ [20].

$$\frac{T_i^{i\in[1,n]} \in \mathcal{T}}{T_i^{i\in[1..n]} \preceq \vee[T_i^{i\in[1,n]}]} \tag{15}$$

Finally, the following rules defines the necessary conditions to be met for a type T to be a supertype of a type $\vee[S_i^{i\in[1,n]}]$.

$$\frac{T \in \mathcal{T} \quad S_i^{i\in[1..n]} \in \mathcal{T} \quad S_i^{i\in[1..n]} \preceq T}{\vee[S_i^{i\in[1..n]}] \preceq T} \tag{16}$$

Again, the opposite rule that defines the premises that must hold so that S is a subtype of a type $\vee[T_i^{i\in[1,n]}]$.

$$\frac{T \in \mathcal{T} \quad S_i^{i\in[1..n]} \in \mathcal{T} \quad S \in \{S_i^{i\in[1..n]}\} \quad T \preceq S}{T \preceq \vee[S_i^{i\in[1..n]}]} \tag{17}$$

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 through 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 objects and ground triples. The poset can be used to compute a join $S \sqcup T$ and a meet $S \sqcap T$. 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. □

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 are 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 $J = \sqcup[T_i^{i\in[1,n]}]$ is a set of MUB types $J_j^{j\in[1,m]} \in J$ such that J_j is an upper bound with $T_i^{i\in[1,n]} \preceq J_j$, and there is no such L where $T_i^{i\in[1,n]} \preceq L$ 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 $T_i^{i\in[1,n]}]$, $M = \sqcap[T_i^{i\in[1,n]}]$, 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 $M_j \preceq T_i^{i\in[1,n]}$, and all other lower bounds U with $U \preceq T_i^{i\in[1,n]}$ 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 $\{T_i^{i\in[1,n]}\}$, the join $J = \sqcup[T_i^{i\in[1,n]}]$ is a set of types $J_j^{j\in[1,m]} \in J$ that are the minimal upper bounds such that $T_i \preceq J_j$ for $i \in [1, n]$. On the other hand, Rule 15 for the $\vee$-types states $T_i \preceq \vee[T_i^{i\in[1,n]}]$. However, the join type and $\vee$-type differ in the interpretation.

$$[\![\vee[T_i^{i\in[1,n]}]]\!]_{\Delta} = \bigcup_{i\in[1..n]} [\![T_i]\!]_{\Delta} \subseteq \bigcup_{j\in[1,m]} [\![J_j]\!]_{\Delta} = [\![\sqcup[T_i^{i\in[1,n]}]]\!]_{\Delta}$$

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

A meet type of $T_i^{i\in[1,n]}$ may not exist in a poset of types from a KG. In general, the meet types $M = \sqcap[T_i^{i\in[1,n]}]$ exist in a class ontology if the types $T_i^{i\in[1,n]}$ 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. A $\wedge$-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 $\wedge[T_i^{i\in[1,n]}]$ denotes the intersection $\bigcap[\![T_i]\!]_\Delta$ while the interpretation of a meet type $M_j \in \sqcap[T_i^{i\in[1,n]}]$ includes the interpretation of the meet types from M. Note that the instances of the meet types are in the intersection of the instances of types $T_i^{i\in[1,n]}$. The set $\bigcap[\![T_i]\!]_\Delta$ can also include objects that are not instances of any meet type from M. Hence,

$$[\![\wedge[T_i^{i\in[1,n]}]]\!]_\Delta = \bigcap_{i\in[1..n]} [\![T_i]\!]_\Delta \supseteq \bigcap_{j\in[1,m]} [\![M_j]\!]_\Delta = [\![\sqcap[T_i^{i\in[1,n]}]]\!]_\Delta.$$

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 5.2. 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

– For $V \in \mathcal{V}$ gather ground types of identifiers with $\wedge$-type as $V :_\downarrow \wedge[T_i^{i\in[1,n]}]$.
– $V \in \mathcal{V} \quad \forall T_i \in \mathcal{T},\ t :_\downarrow T_i$.
– The ground type of V is a type $T_g = \wedge[T_i^{i\in[1,n]}]$.
– The following rule geathers all ground types of $V \in \mathcal{V}$.

$$\frac{V \in \mathcal{V} \quad T_i^{i\in[1,n]} \in \mathcal{T} \quad V :_\downarrow T_i^{i\in[1,n]}}{V :_\downarrow \wedge[T_i^{i\in[1,n]}]} \tag{18}$$

Let's have a look at $\wedge$ type composed of V's ground types $T_i^{i\in[1,n]}$ in the case $V \in \mathcal{I}_i$. In Yago [11], often V has a set of very specific classes C_s but also some general classes C_g. The general classes C_g are close to the classes used in the stored triple types. If stored typing of V is correct, then C_g includes the classes that are supertypes of classes from C_s.

The ground type $\wedge[T_i^{i\in[1,n]}]$ 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 $\{T_i^{i\in[1,n]}\}$ with respect to $\preceq$ [6]. The super-types of the minimal elements of $\{T_i^{i\in[1,n]}\}$ include all valid types of V. Hence, we use the set of minimal elements from $\{T_i^{i\in[1,n]}\}$ as the starting point to explore the relations between the ground triple types and the user-defined triple types of a triple t including V as a component.

The operator MIN is defined on a poset of types $(\{T_i^{i\in[1,n]}\}, \preceq)$. Given a set of types $\{T_i^{i\in[1..n]}\}$ the MIN operator retains types $S_j^{j\in[1,m]} \in \{T_i^{i\in[1,n]}\}$ such that $\nexists T_k^{k\in[1,n]}(T_k \prec S_j)$. All pairs of types $S_k, S_l \in \{S_j^{j\in[1,m]}\}$ with $k \neq l$ are *incomparable*, i.e.,

$S_1 \not\sim S_2 \equiv S_1 \not\preceq S_2 \wedge S_1 \not\succeq S_2$. The logical rule for the operation MIN is as follows.

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

The rule says that S is a minimal type of a ground type $\wedge[T_i^{i\in[1,n]}]$. S is minimal since all other T_i types are either more general or equal ($\succeq$), or not related to S. The rule generates all MIN types of $\wedge[T_i^{i\in[1,n]}]$.

The following rule is used for gathering all MIN types of $\wedge[T_i^{i\in[1,n]}]$. The result is a conjunction of minimal types $\wedge[S_1..S_m]$.

$$\frac{V \in \mathcal{V} \quad \forall i \in [1,m],\ V :_{\Downarrow} S_i}{V :_{\Downarrow} \wedge[S_j^{j\in[1,m]}]} \tag{20}$$

Now we have a minimal ground type of a value V in the form of a conjunction of minimal types S_i of V. The types that are important from the perspective of typing computer languages defined on values from a KG are the user-defined types of triples. The definition of a user-defined triple type requires the knowledge about the meaning of the binary relationship defined by a predicate—this is reflected in the selection of the domain and range of the predicate.

The first step in verifying the relations between the ground and user-defined types of a value V is the computation of a join type from a ground type of V. The join type of a ground type should be a subtype of the user-defined types that describe the value V. If the join type is not a subtype of the user-defined type then there is an error in stored types of a value V.

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

Rules 21-22 present the logical definition of the join type. The following Rule 21 determines one join type but can be used to generate all join types.

$$\frac{V \in \mathcal{V},\ V :_{\downarrow} \wedge[T_i^{i\in[1,n]}] \quad S \in \mathcal{T},\ T_i^{i\in[1,n]} \preceq S \quad \forall P \in \mathcal{T},\ (T_i^{i\in[1,n]} \preceq P \wedge S \preceq P) \vee P \not\sim S}{V :_{\sqcup} S} \tag{21}$$

The individual join types derived by the above rule are gathered into one $\wedge$ type of join types by using the following rule.

$$\frac{V \in \mathcal{V} \quad S_i^{i\in[1,m]} \in \mathcal{T} \quad \forall i \in [1,m],\ V :_{\sqcup} S_i}{V :_{\sqcup} \wedge[S_i^{i\in[1,m]}]} \tag{22}$$

The join types are used as the starting point for searching the correct user-defined type. In the case the predicate of has two different definitions in two different contextes,

then the paths from join types to the domain and range classes determines the definition of the triple type. The details are presented in Section 5.3.

5 Typing triples

5.1 Triple types

A type of a triple $(s, p, o) \in \mathcal{V}$ is a triple $(D, p, R) \in \mathcal{T}$ such that $s : D$ and $o : R$ holds. The types D and R are type expressions that represent either a class or a $\wedge$-type composed of classes.

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

$$(\wedge[S_i^{i\in[1,n]}], p, \wedge[O_j^{j\in[1,m]}]).$$

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

$$\frac{T_1 \in \mathcal{T},\ T_1 = (D_1, p_1, R_1) \quad T_2 \in \mathcal{T},\ T_2 = (D_2, p_2, R_2) \quad D_1 \preceq D_2 \quad p_1 \preceq p_2 \quad R_1 \preceq R_2}{T_1 \preceq T_2} \tag{23}$$

It is obvious from the definition of the subtyping relationship among the triple types that $[\![T_1]\!]_{\Delta} \subseteq [\![T_2]\!]_{\Delta}$. Note that Rule 23 handles the $\wedge$-types of the subject and object through Rules 12-14.

5.2 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 $T_s * p * T_o$ that we represent as a triple (T_s, p, T_o). 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.

$$\frac{t \in \mathcal{V},\ t = (I_s, p, I_o) \quad T_s, T_o \in \mathcal{T} \quad I_s :_{\downarrow} T_s \quad I_o :_{\downarrow} T_o \quad p :_{\downarrow} \text{rdf:Property}}{t :_{\downarrow} (T_s, p, T_o)} \tag{24}$$

The type T_s 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.

$$\frac{t \in \mathcal{V},\ t = (I_s, p, I_o) \quad T_s, T_o \in \mathcal{T} \quad I_s :_{\Downarrow} T_s \quad I_o :_{\Downarrow} T_o \quad p :_{\downarrow} \text{rdf:Property}}{t :_{\downarrow} (T_s, p, T_o)} \tag{25}$$

The component types T_s and T_o of the minimal ground type (T_s, p, T_o) 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.

$$\frac{t \in \mathcal{V} \quad t : (\wedge[S_i^{i\in[1,n]}], p, \wedge[R_j^{j\in[1,m]}])}{t : \bigwedge_{i\in[1,n],j\in[1,m]}[(S_i, p, T_j)]} \tag{26}$$

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 stored triple types to select the valid stored triple type of a triple. This is detailed in Section 5.3.

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.

$$\frac{t \in \mathcal{V},\ t = (I_s, p, I_o) \quad T_s, T_o \in \mathcal{T} \quad I_s :_{\sqcup} T_s \quad I_o :_{\sqcup} T_o \quad p :_{\downarrow} \text{rdf:Property}}{t :_{\downarrow} (T_s, p, T_o)} \tag{27}$$

The join type (T_s, p, T_o) 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 27 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.

5.3 Stored triple types

The stored triple types are types of triples defined by RDF-Schema [25] vocabulary. Each predicate 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 is defined for p's super-predicates and inherited to the predicate p.

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.

Let us now inspect the same problem from the perspecive of predicate modelling. In the case one predicate stands for one sense of a predicate, a polysemic predicate is specialized to sub-predicates modelling different senses. In the case that predicates of a KG have many senses, this representation can lead to predicate explosion. However, the reasoning is simpler if all predicates have unique meaning. The use of one sense for one predicate is detailed in Section 5.3.1.

The second way of modelling predicate sense is definition of the same predicate in multiple contexts. The context can be defined in many ways. Examples of contexts in KGs are RDF named graphs [5], microtheories of Cyc [23], and spaces in Scone [8]. The corresponding reasoners of KGs can define and use contextes. A predicate can be defined differently in different contexts but within one context we can only have one meaning of a predicate.

The context (sense) of a predicate can also be defined by using triple types including a given predicate. In other words, a sense of a predicate depends on the context implemented by a triple type. The definition of one predicate in multiple contexts is studied in 5.3.2.

5.3.1 KGs with RDF-Schema. Let us first define the rules for finding a stored type of a triple in the case RDF-Schema semantics is used in a KG. We first determine all stored triple types for a given triple $t = (s, p, o)$. A stored type is constructed by selecting triple types that are composed of 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 $p' \succ p$.

$$\frac{\begin{array}{c} t \in \mathcal{V},\, t = (s,p,o) \quad p_1, p_2 \in \mathcal{T},\, p \preceq p_1 \preceq p_2 \\ T_i^{i\in[1,n]} \in \mathcal{T},\, (p_1, \text{domain}, T_i) \in \Delta \quad S_j^{j\in[1,m]} \in \mathcal{T},\, (p_2, \text{range}, S_j) \in \Delta \end{array}}{t :_{\uparrow} \wedge[T_i^{i\in[1,n]}] * p * \wedge[S_j^{j\in[1,m]}]} \tag{28}$$

The above rule generates the types of the domain and range of a predicate p. The valid domain or range types of the predicate p can be any of types defined as domain or range for some $p' \succeq p$. (****) Note that domain and range types can be defined for different predicates p_1 and p_2 "above" p ($p_1, p_2 \succeq p$) which have to be comparable, i.e., $p_1 \sim p_2$. 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. In the second part of the premise the domain and range types are gathered for the selected predicates p_1 and p_2, respectively.

The above Rule 28 generates all valid stored 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 stored triple type of a t.

$$\frac{t \in \mathcal{T}_i \quad T \in \mathcal{T}, t :_{\uparrow} T \quad S_i^{i \in [1,n]} \in \mathcal{T}, t :_{\uparrow} S_i \quad T \preceq S_i \vee T \not\sim S_i}{t :_{\uparrow} T} \tag{29}$$

The first part of 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 $S_i^{i \in [1,n]}$ 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 stored typing of a KG (using RDF.Schema) is correct then the condition $T \sim S_i$ is always $true$.

In the case that our KG is restricted to RDF-Schema data model then one predicate must represent exactly one meaning. Therefore, Rule 29 generates exactly one minimal type. Furthermore, under the restrictions of RDF-Schema we can not define a predicate p with two meanings. If we would specify two different domains or ranges of p then reasoner would treat the domain and range types as $\wedge$-types. Each instance of the domain (range) type have to be an instance of all specified types of the domain (range).

5.3.2 KGs with the contextual representation. The collective findings of the research in the area of Cognitive science [12] show 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 predicates in KGs, the contextual representation and reasoning allows the definition of multiple senses of a predicate. There are many motivations for adopting contextual representation and reasoning in a KG. First, with the evolvment of KGs there are many examples where a KG is represented in the modular way spliting 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], Wiki Data [30] and Yago [11]. Another example is Cyc [23] that uses microtheories to represent different contexts. Similarly 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 mutiple 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 explicitely 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 function 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 the case there are more than one triple types including a predicate p then these are treated as alternatives. For example, if a KG includes triples (c_1, p, c_2) and (c_3, p, c_4), where $c_i \in \mathcal{I}_c$, then the type of ground triples including p is $c_1 * p * c_2 \vee c_3 * p * c_4$.

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

$$\wedge[c_i^{i\in[1,n]}] * p * \wedge[c_j^{j\in[1,m]}].$$

Let us now present the rules which 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 30 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.

$$\frac{t \in \mathcal{V},\, t = (s, p, o) \quad T_i^{i\in[1,n]} \in \mathcal{T},\, T_i = (D_i, p, R_i) \quad T_i^{i\in[1,n]} \in \Delta}{t :_\uparrow \vee[T_i^{i\in[1,n]}]} \tag{30}$$

A stored triple type $\vee[T_i^{i\in[1,n]}]$ of t is a disjunction of all valid stored triple types of t. Similarly to Rules 19-20, which are defined to find minimal types of a set of classes, the following Rules 31-32 generate the set of minimal stored triple types of a set including all valid stored tyes of t.

$$\frac{t \in \mathcal{V} \quad t :_\uparrow \vee[T_i^{i\in[1,n]}] \quad S \in \{T_k^{k\in[1,n]}\} \quad \forall i \in [1,n],\, S \preceq T_i \vee S \not\sim T_i}{t :_\uparrow S} \tag{31}$$

The rule says that S is a minimal stored triple type of a stored triple type $\vee[T_i^{i\in[1,n]}]$. S is minimal since all other types T_i are either more general or equal ($\succeq$), or not related to S. The following rule gathers all minimal stored triple types of $\vee[T_i^{i\in[1,n]}]$.

[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.

$$\frac{t \in \mathcal{V} \quad \forall i \in [1,m],\ t :_{\uparrow} S_i}{t :_{\uparrow} \vee[S_j^{j \in [1,m]}]} \tag{32}$$

The above rule is identical to the Rule 20 except that in this settings it handles triples and not identifiers. The result is a disjunction of minimal stored triple types $S_j^{j \in [1,m]}$.

5.4 Typing a triple

...

The type of a triple $t = (s, p, o)$ is computed by first deriving the base type T and the top type S of t. Then, we check if S is reachable from T through the sub-class and sub-property hierarchies, i.e., $T \preceq S$.

$$\frac{t \in \mathcal{T}_i \quad T \in \mathcal{T}_t,\ t :_{\downarrow} T \quad S \in \mathcal{T}_t,\ t :_{\uparrow} S \quad T \preceq S}{t : S} \tag{33}$$

...

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, can be used in tasks such as query optimization and reasoning. Finally, we show in the paper how types can be employed to disambiguate the sense of a predicate.

...

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.