Type-checking knowledge graphs

Iztok Savnik[1]

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

Abstract. This is an abstract...

Keywords: type checking · knowledge graphs · RDF stores · graph databases.

Table of Contents

1 Introduction

– Introduction to knowledge graphs (KG)... [7, 4].
– KGs are becoming knowledge bases (KB)...
– What are the structural characteristics of KBs?
– What KBs can represent that (classical) data models can not.
– Relations between the knowledge bases and KGs.

– In KGs we have types of individual objects represented as classes.
– Further, the types of the triples are the triples including types of individual objects.
– Types represent a higher-level description of ground triples.
– Types can be used to verify the correctness of the ground triples and the structures that they form.
– Types define the context in a KG that allows placing (?) a structure of triples (sub-graph) in a KG.
– Disambiguation of property (predicate) placement. Later, binding of methods, etc.

– On the type theory... [10, 5].
– Classical type-checking problem in programming languages.
– On differences of type-checking KGs to classical type-checking problem.
– On intersection and union types and their use in type-checking KGs [10, 1, 8].

– KG domain is complex because KG is a knowledge representation (KR) language.
– We have a specific domain, i.e., a knowledge graph including nodes and triples.
– Abstract insight into the structure of KG.
– The classes form an ontology that can formally be represented as a poset.
– Consequently, triple types are also ordered in a poset.
– Denotational view of classes and type triples.
– The interpretations of classes and triple types form a poset based on the subset relation.

– Type-checking of KGs (abstract).
– Type-checking of ground triples from a KG.
– Three phases of type-checking ground triples.
– First, a lub type T_{lub} *of a ground triple is derived,*
– Second, a glb type T_{glb} *of the stored triple types is computed.*
– Finally, a sub-type relationship between the types T_{lub} *and* T_{glb} *is investigated.*
– The type T_{lub} *restricts* T_{glb} *in cases that the propery of* t *has multiple different meanings.*
– Typing triple patterns and BGP queries in further work.
– Identifying errors in typing of a KG.

– The problem is in between type checking and type inference.
– Using stored types of idents to infer the type of an object (ground triple) and then check how it relates to stored types of triples.
– The idea is close to bideriectional typing [3] because of inferring and checking.
– In KGs we first infer as much as possible and then check inferred type with the stored types.

–

2 Preliminaries

2.1 Knowledge graph

This section defines a knowledge graph as a RDF graph [11] using RDF-Schema [12] for the representation of the structural part of a knowledge base.

Let I be the set of URI-s, B be the set of blanks and L be the 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.

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. In view of the above formal representation of RDF triples, the complete set of identifiers is $\mathcal{I} = I \cup B \cup L$. The identifiers from $\mathcal{I}$ are classified into the sets including literals $\mathcal{I}_l$, individual (ground) identifiers $\mathcal{I}_i$, class identifiers $\mathcal{I}_c$, predicate identifiers $\mathcal{I}_p$.

The complete set of triples, referred to as $\mathcal{T}$, is classified into the sets of individual (ground) triples $\mathcal{T}_i$, triple types $\mathcal{T}_t$ and abstract triples $\mathcal{T}_a$. The individual triples include solely the individual identifiers $\mathcal{I}_i$ and predicates $\mathcal{I}_p$. The triple types include only class identifiers $\mathcal{I}_c$ and predicates. Finally, the abstract triples link individual and class identifiers in single triples.

2.2 Typing rule language

In this paper we define typing of a data language used to represent a TBOX [2] of a knowledge base given in a form of a knowledge graph. The data language specifys the assertions in the form of triples and the schema of assertions as the types of triples. The ground triples are the instances of the triple types that altogether define the schema of a KG.

In comparison to the data structures used in programming languages [10, 5], the data language of a KG is more complex. First, a KG graph includes an ontology of classes and properties. Second, typing of ground identifiers is stored in a KG, i.e., each ground identifier has one or more types represented as class identifiers. Further, the properties (predicates) of a KG are treated as objects that are included in a classification hierarchy of properties. For each property we have the definition of one or more triple types stored in a KG. Finally, the properties (including triple types) are inherited through the classification hierarchy of classes and predicates.

Furthermore, the data language of a KG, which is based on RDF and RDF-Schema [11, 12], does not include variables as in the case for the expressions of a programming language. All information needed for typing a ground triple is available from a KG.

For the above presented reasons, we do not use standard typing rule language [10, 5] that includes the context Γ where the types of the variables are stored. We use more expressive meta-language that is rooted in first order logic (abbr. FOL). The rules are composed of a set of premises and a conclusion. The premises are either typing judgements in the form $o : T$, or expressions in the FOL. 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 jufgement.

The rules are grounded in a knowledge graph through the sets of identifiers and triples that are defined in Section 2.1. In rules we specify the domain for each symbol used. 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.

Similar to [3], we differ between two interpretations of rules. First, the *generator* view of rules is the forward interpretation where rules sythesize the types from the types derived by premises. 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. Second, the *type-checking* view of the rules is the backward interpretation. Given the symbol and its type, the construction of a given type is checked by the rules.

3 Typing identifiers

The set of identifiers $\mathcal{I}$ include ground identifiers $\mathcal{I}_g$, class identifiers $\mathcal{I}_c$, and the predicates (properties) $\mathcal{I}_p$ that are are both ground identifiers, since they are instances of rdf:Property, and similar to class identifiers, since they act as types and form an ontology of predicates.

In this section we present typing of ground identifiers. The types of ground identifiers are further used for typing ground triples in Section 4. However, before presenting the types of identifiers, we introduce the intersection and union types that are used for the description of the types of identifiers in the following Section 3.4. Typing of literals 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 and the complete types of ground identifiers are presented in Section 3.3.

3.1 Typing literals

Literals are the values of an atomic type. The atomic types are in RDF provided by the RDF-Schema dictionary [12]. RDF-Schema defines a list of atomic types, such as xsd:integer, xsd:string, or xsd:boolean.

Typing of the atomic types is determined by the following rule.

$$\frac{L \in \mathcal{I}_l \quad T \in \mathcal{I}_c, \quad \text{"L"\^{}\^{}T} \in \mathcal{L}}{L : T} \tag{1}$$

The rule states that a literal value L is of a type T if a literal "L"^^T is an element of the set of literals $\mathcal{L}$. A literal "L"^^T includes a literal value L and a literal type T referencing a type from the RDF-Schema dictionary. As an example, the literal "365"^^xsd:integer includes the literal value 365 and its type xsd:integer.

3.2 Stored typing and subtyping of identifiers

The expression $I :_{\downarrow} C$ states that a class C is a type of an individual identifier I. The expression $I_1 \preceq_{\downarrow} I_2$ defines the subtype (sub-class) relationship between the class identifiers I_1 and I_2. The index '$\downarrow$' in relations $:_{\downarrow}$ and $\preceq_{\downarrow}$ denotes that the relations are stored in a database—we refere to them as *one-step* typing and subtyping relations. Such notation allows us to address differently the *stored* and the *derived* types of the graph database schema.

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

$$\frac{I \in \mathcal{I}_i \quad I_c \in \mathcal{I}_c \quad (I, \text{rdf:type}, I_c) \in \mathcal{D}}{I :_{\downarrow} I_c} \tag{2}$$

The individual entity I can have more than one stored types. By using a generative interpretation, Rule 2 synthesizes all types I_c such that $I : I_c$. The rule can be used either in some other rule that employ 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 3.4.

A one-step subtyping relationship $\preceq_{\downarrow}$ is defined by means of the RDF predicate rdfs:subClassOf in the following rule.

$$\frac{I_1, I_2 \in \mathcal{I}_c \quad (I_1, \text{rdfs:subClassOf}, I_2) \in \mathcal{D}}{I_1 \preceq_{\downarrow} I_2} \tag{3}$$

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

$$\frac{I_1, I_2 \in \mathcal{I}_p \quad (I_1, \text{rdfs:subPropertyOf}, I_2) \in \mathcal{D}}{I_1 \preceq_{\downarrow} I_2} \tag{4}$$

3.3 Typing and subtyping identifiers

The one-step relationship $\preceq_{\downarrow}$ is now extended with the reflectivity, transitivity and asymmetry to obtain the relationship $\preceq$. 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 $\mathcal{I}_c$.

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

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

$$\frac{I_c \in \mathcal{I}_c}{I_c \preceq I_c} \tag{6}$$

The subtype relationship is also transitive.

$$\frac{I_1, I_2, I_3 \in \mathcal{I}_c \quad I_1 \preceq I_2 \quad I_2 \preceq I_3}{I_1 \preceq I_3} \tag{7}$$

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

$$\frac{I_1, I_2 \in \mathcal{I}_c \quad I_1 \preceq I_2 \quad I_2 \preceq I_1}{I_1 = I_2} \tag{8}$$

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 [6]. The following rule specifys that all class identifiers are more specific than $\top$.

$$\frac{\forall S \in \mathcal{I}_c}{S \preceq \top} \tag{9}$$

The stored 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.

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

The link between the typing relation and subtype relation is provided by adding a typing rule called *rule of subsumption* [10].

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

3.4 Intersection and union types

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, $\wedge[T_1 \ldots T_n]$ is the greatest lower bound (abbr. GLB) of types $T_1 \ldots T_n$ [8, 9]. The instances of the type $\wedge[T_1 \ldots T_n]$ 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 $\tau = \mathcal{I}_c \cup \mathcal{T}_t$ is used to ground the types in the rules.

The instances of a type $\wedge[T_1..T_n]$ are the instances of all particular types T_i. This is stated by the following rule.

$$\frac{\forall i \in [1..n],\ T_i \in \tau}{\wedge[T_1..T_n] \preceq T_i} \tag{12}$$

Further, the following rule states that if the type S is more specific than the types $T_1, \ldots, T_n$ then S is more specific then $\wedge[T_1..T_n]$.

$$\frac{S \in \tau \quad \forall i \in [1..n],\ T_i \in \tau \quad S \preceq T_i}{S \preceq \wedge[T_1..T_n]} \tag{13}$$

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, $\vee[T_1 \dots T_n]$ is the least upper bound of types $T_1, \dots, T_n$ [8].

$$\frac{\forall i \in [1..n],\ T_i \in \tau}{T_i \preceq \vee[T_1..T_n]} \tag{14}$$

Finally, if the type T is more general than the types $S_1, \dots, S_n$ then T is more general then $\vee[S_1 \dots S_n]$.

$$\frac{T \in \tau \quad \forall i \in [1..n],\ S_i \in \tau \quad S_i \preceq T}{\vee[S_1 \dots S_n] \preceq T} \tag{15}$$

Semantics of $\wedge$ and $\vee$ types in KGs. The meaning of the $\wedge$ and $\vee$ types can be defined through their interpretations. The following definition expresses the denotation of a $\vee$ type with the interpretations of its component types. Suppose we have a set of types $\forall i \in [1..n],\ T_i \in \tau$.

$$[\![\vee[T_1..T_n]]\!]_{\mathcal{D}} = \bigcup_{i \in [1..n]} [\![T_i]\!]_{\mathcal{D}}$$

Similarly, the interpretation of a $\wedge$ type is the intersection of the interpretations of its component types.

$$[\![\wedge[T_1..T_n]]\!]_{\mathcal{D}} = \bigcap_{i \in [1..n]} [\![T_i]\!]_{\mathcal{D}}$$

Note that the interpretation of a class C is a set of instances $[\![C]\!]_{\mathcal{D}} = \{I \mid I \in \mathcal{I}_i \wedge I : C\}$. Further, the interpretation of a triple type T is the set of ground triples $[\![T]\!]_{\mathcal{D}} = \{t \mid t \in \mathcal{T}_t \wedge t : T\}$[1][14].

3.5 The join and meet types

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

A KG includes a stored poset of classes that represent types of the individual objects. The poset can be used to compute a join type $S \sqcap T$ and a meet type $S \sqcup T$ [10]. The join type $J = S \sqcap T$ is the least type such that $S \preceq J$, $T \preceq J$, i.e., for all types U, if $S \preceq U$ and $T \preceq U$, then $J \preceq U$.

The meet type $M = S \sqcup T$ is the greatest type such that $S \preceq M$, $T \preceq M$ and there is no such L where $S \preceq L$ and $T \preceq L$ without also having $L \preceq M$. Since we have a top type $\top$ defined in a KG, the join of arbitrary two types always exists. However, the meet of two arbitrary types may not exist always.

[1] Triple types $\mathcal{T}_t$ are presented in the following Section 4.

The join type is related to the $\vee$-type. Given a set of types $\{T_1 \dots T_n\}$, the join type $T = \sqcup[T_1..T_n]$ represents a LUB type such that $T_i \preceq T$ for all i. On the other hand, Rule 14 for the $\vee$-types states $T_i \preceq \vee[T_1..T_n]$. However, the join type and $\vee$-type differ in the interpretation.

$$\llbracket \vee[T_1..T_n] \rrbracket_{\mathcal{D}} = \bigcup_{i \in [1..n]} \rrbracket T_i \rrbracket_{\mathcal{D}} \subseteq \llbracket T \rrbracket_{\mathcal{D}} = \llbracket \sqcup[T_1..T_n] \rrbracket_{\mathcal{D}}$$

While the interpretation of the type $\vee[T_1..T_n]$ includes precisely the instances of all T_i, the type T is a LUB class and the interpretation of $T = \sqcup[T_1..T_n]$ can include interpretations of classes that are not among $[T_1..T_n]$.

A meet type of $[T_1..T_n]$ may not exist in a poset of classes from a KG. In general, a meet type $M = \sqcap[T_1..T_n]$ exists in a class ontology if the types $T_1..T_n$ are *bounded below* which means that there exists a type L such that $L \preceq T_i$ for all i. The bounded meet types [10] are not frequent on the lower levels of a class ontology from a KG.

Similarly to the $\vee$-type and the join type, the semantics of the $\wedge$-type is similar to the semantics of meet type. Both of them define a GLB type. The type $\wedge[T_1..T_n]$ denotes the intersection $\bigcap \llbracket T_i \rrbracket_{\mathcal{D}}$ while the interpretation of a meet type $M = \sqcap[T_1..T_n]$ includes solely the interpretation of M. The set $\bigcap \llbracket T_i \rrbracket_{\mathcal{D}}$ can also include objects that are not instances of M. Hence,

$$\llbracket \sqcap[T_1..T_n] \rrbracket_{\mathcal{D}} = \llbracket M \rrbracket_{\mathcal{D}} \subseteq \llbracket \wedge[T_1..T_n] \rrbracket_{\mathcal{D}} = \bigcap_{i \in [1..n]} \llbracket T_i \rrbracket_{\mathcal{D}}.$$

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 4.2. The join as well as meet types are very useful in the procedure for type-checking basic graph patterns [13]. 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 type of a GP variable.

3.6 Typing identifiers with $\wedge$ and $\vee$ types

– *The use of $\wedge$ and $\vee$ types to describe identifiers.*

– *For $I \in \mathcal{I}_i$ gather ground types of identifiers with $\wedge$-type as $I :_{\downarrow} \wedge[S_1..S_n]$.*
  – *$I \in \mathcal{I}_i \quad \forall S_i \in \mathcal{I}_c,\ t :_{\downarrow} S_i$.*
  – *The ground type of I is a type $S_g = \wedge[S_1..S_n]$.*
  – *The following rule geathers all ground types of $I \in \mathcal{I}_i$.*

$$\frac{I \in \mathcal{I}_i \quad \forall i \in [1..n],\ I :_{\downarrow} S_i}{I :_{\downarrow} \wedge[S_1..S_n]} \tag{16}$$

– *Let's have a look at $\wedge$ types of I's ground types $S_1..S_n$ from a KG.*
  – *Often I 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 I is correct, then s_s have to include subclasses of s_g.*
– *Rule for filtering $\wedge[S_1..S_n]$ all $S_i \preceq S_k$ by using algorithmic typing.*

$$\frac{I \in \mathcal{I}_i \quad I :_{\downarrow} \wedge[S_1..S_n] \quad \forall i \in [1,n],\ R_j \in \{S_k^{k\in[1,n]}\},\ S_i \preceq R_j}{I :_{\downarrow} \wedge[R_1..R_m]} \tag{17}$$

– *For $I \in \mathcal{I}_i$ compute a join type of $S_1..S_n$ as $I : \sqcup[S_1..S_n]$.*
– *A join type $S = \sqcup[S_1..S_n]$ is the LUB type of $[S_1..S_n]$.*
– $[\![\wedge[S_1..S_n]]\!] \subseteq \bigcup_{i\in[1..n]} [\![S_i]\!] \subseteq [\![S]\!]$.

$$\frac{I \in \mathcal{I}_i \quad \forall i \in [1..n],\ I :_{\downarrow} S_i \quad S = \sqcup[S_1..S_n]}{I :_{\sqcup} S} \tag{18}$$

– *Let's have a look at $\sqcup$ types of I's ground types $S_1..S_n$ from a KG.*
– *In many cases I has a single type S_1 which is the same as the join type S.*
– *The super-classes of the join type S have to be included in the stored triple types $\mathcal{T}_t$*
– *to be defined as the types of subject or object.*
– *Often join type S is close to the classes that are components of stored triple types.*

– *Why computing a join type $S = \sqcup[S_1..S_n]$ of $\wedge[S_1..S_m]$?*
– *Show how the $\wedge$ and $\sqcup$ types of I are used to compute a type of I.*
– *$t :_{\sqcup} S$ and there should be a path from S to class components of stored triple types T.*
– *... details about the above statement.*
– *As such, S is an approprite point to start searching stored types of t.*
–

– *In a class poset we can have more than one $\sqcup$ types of $[S_1..S_n]$.*
– *All $\sqcup$ types of a given set of types $\{S_1, ..., S_n\}$.*
– *Show how all $\sqcup$ types can be gathered into one $\wedge$ or $\vee$ type.*
– *Show how join types of $\wedge$ type are gathered with $\wedge$ type of join types.*
– *... the same for $\vee$ types.*

4 Typing triples

– *We would like to type of a triple $t \in \mathcal{T}_i$.*
– *Abstract of the procedure.*
– *We compute first the ground type $T_g = \wedge[T_1..T_m]$ and a stored type T_s of t.*
– *The ground type T_g is computed from the ground types of t's components.*
– *The subtype relation should hold $T_g \preceq T_s$.*

– *There are two basic aspects of a triple type.*
– *First, the type is computed bottom-up: from the stored types of triple components.*
– *Second, the type can be computed top-down: from the user-defined domain/range types of properties.*

– *About the types that are computed bottom-up.*
– *Ground type of a triple is computed first using* $:_\downarrow$*.*
– *Next, the lub type of a triple is derived using* $:_\sqcup$*.*

– *About the stored types that are computed as glb of valid stored types.*
– *From the top side of the ontology, the stored type* $:_\uparrow$ *is determined based on* p*.*
– *The glb types of all types obtained using* $:_\uparrow$ *obtaining a glb type* $:_\sqcap$*.*
– *Finally, the type* $:$ *of* t *is deterimned by summing alternative* $:_\uparrow$ *types.*

– *Interactions between the* $\wedge/\vee$ *types of triple components and triples must be added.*
– *Analogy between the types of functions in LC and types of triples.*
– *Show rules relating* $\wedge/\vee$ *types and triple types. Example.*
– *E.g.,* $(S_1 \wedge S_2) * p * R = S_1 * p * R \wedge S_2 * p * R$*.*
– *Are all rules covered?*

– *Predicates should be treated in the same way as the classes.*
– *They can have a rich hierarchy.*
– Note: *Where to include discussion on special role of predicates and their relations to classes?*
– *Mention Cyc as the practical KB with rich hierarchy of predicates.*

4.1 Product types

4.2 Deriving a ground type of a triple

A ground type of an individual identifier i is a class C related to i by one-step type relationship $:_\downarrow$ denoting a ground type. 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 triple $T = C_s * p * C_o$ that 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.

$$\frac{t \in \mathcal{T}_i,\ t = (I_s, p, I_o) \quad I_s :_\downarrow C_s \quad I_o :_\downarrow C_o \quad p :_\downarrow \text{rdf:Property}}{t :_\downarrow C_s * p * C_o} \tag{19}$$

The class C_s is one of the ground types of I_s, and the type C_o is one of the ground types of I_o. The predicates are treated differently to the subject and object components of triples. The predicates have the role of classes while they are instances of rdf:Property.

There can be multiple ground types of a triple. They may be gathered into a single $\wedge$-type by using the following rule. The types $T_1,\ldots,T_n$ are obtained using Rule 19.

$$\frac{\forall i \in [1..n],\ t :_\downarrow T_i}{t :_\downarrow \wedge[T_i]} \tag{20}$$

– *Typing using lub types of* $T_1..T_n$*. Explain why this is needed?*

Let us now define the least upper bound types (abbr. *lub*) of ground types derived by Rule 20. Since a partialy ordered set is not a lattice, we can have more than one lub type for a given set of ground types.

The lub types of a given list of triple types $T_1..T_n$ are computed in two steps as before when gathering multiple ground types with conjunction. A single lub type is defined as follows.

$$\frac{t :_{\downarrow} \wedge[T_1..T_n] \quad T \in \mathcal{T}_t, \quad \forall i,\ T_i \preceq T \quad \forall S \in \mathcal{T}_t,\ \forall i,\ (T_i \preceq S \wedge T \preceq S) \vee T \not\succ S}{t :_{\sqcup} T} \tag{21}$$

$$\frac{t :_{\downarrow} \wedge[T_1..T_n] \quad \mapsto T = \sqcup[T_1..T_n]}{t :_{\downarrow\sqcup} T} \tag{22}$$

$$\frac{t :_{\downarrow} \wedge[T_1..T_n] \quad \mapsto \forall i \in [1..m],\ S_i = \sqcup[T_1..T_n]}{t :_{\downarrow\sqcup} \wedge[S_1..S_m]} \tag{23}$$

The above rule states that a type T is a lub type of a ground type $\wedge[T_1..T_n]$ if all ground types T_i are subtypes of T. Furthermore, the lub type T is the least (closest) supertype of all members of ground $\wedge$-type $T_1, \ldots, T_n$. The lub types can be now gathered using the following rule.

$$\frac{\forall i \in [1..n](t :_{\sqcup} T_i)}{t :_{\sqcup} \wedge[T_1..T_n]} \tag{24}$$

4.3 Stored types of triples

– *General comments.*
– *Analysis tool. Show minimality of the stored types (either enumerated or gathered with $\bigvee$).*
– Reminder: *when a complete stored (user-defined) type is related to the base type of a triple, some of GLB types may be eliminated.*

– *Present the complete story.*
– *Computing the minimal and valid stored type of a triple $t = (s, p, o) \in \mathcal{T}_i$.*
– *Stored types are defined by linking a predicate p to a domain and range classes.*
– *Only types (domains and ranges) defined for $p' \succeq p$ are valid stored types.*
– *There are no other valid types below, i.e., for $p' \prec p$.*
– *Among the valid stored types the most specific and unrelated stored types are selected.*
– *In other words, only glb types of valid stored types are selected.*
– *Finally, the minimal and complete type of t is an $\vee$-type including all previously selected glb types.*

We first find stored triple types for a given triple $t = (s, p, o)$. A stored schema triple is constructed by selecting types including a predicates $p' \succeq p$ that the domain and range defined.

$$\frac{t \in \mathcal{T}_i,\, t = (s,p,o) \quad p' \in \mathcal{I}_p, p \preceq p' \quad (p', \text{domain}, T_s) \in g \quad (p', \text{range}, T_o) \in g}{t :_{\uparrow} T_s * p * T_o} \tag{25}$$

– Comments and description of the above rule.
– Note p is used for all types. p shoud be in most specific type -
– It makes no sense to generate types with p'.

The domain and range of a predicate p can be defined for any super-predicate, they do not need to be defined particulary for p. In addition, the domain and range of a predicates do not need to be defined for the same predicate; they can be defined for any of the super-predicates separately. The following rule captures also the last statement.

– Somewhere here, the inheritance should be noted.
– Inheritance should be treated in knowledge graphs!
– Predicates inherit in the same way as the classes.

$$\frac{t \in \mathcal{T}_i, t = (s,p,o) \quad p_1, p_2 \in \mathcal{I}_p \quad p \preceq p_1 \quad p \preceq p_2 \quad (p_1, \text{domain}, T_s) \in g \quad (p_2, \text{range}, T_o) \in g}{t :_{\uparrow} T_s * p * T_o} \tag{26}$$

– Explanation of the rule.
– If p inherits from multiple $p' \succeq p$, then the above rule generates multiple types. Explain.
– Note that the type is determined only if the domain and range of p or some $p' \succeq p$ is defined.
– Otherwise, the domain and range shoud be $\top$. This should be included.

The following rule is a judgment for a (user-defined) type of a concrete triple $t = (s,p,o)$. A user-defined type of t is the greatest lower bound (abbr. glb) of stored types generated by the rule 25.

– Valid stored types of t: the smallest valid glb types of all stored types.
– Justification: smallest interpretation - smallest search space for queries.
– Valid stored types are solely those defined "above" p.
– The glb types of valid stored types "above" p is selected!
– The rule generates one glb type by one.
– These (glb types) are collected in a $\vee$-type including all GLB types.

*– The meaning of $\not\sim$ is "*not related*".*
– This can be either that we have two p roots with unrelated glb schema triples (trees up).
– Or, two p-rooted but unrelated stored types through multiple inheritance.
– Therefore, we can have more than one stored GLB types.

$$\frac{t \in \mathcal{T}_i \quad T \in \mathcal{T}_t,\, t :_{\uparrow} T \quad \forall S \in \mathcal{T}_t,\, t :_{\uparrow} S \quad T \preceq S \vee T_i \not\sim S}{t :_{\bar{\uparrow}} T} \tag{27}$$

The first premise says that t is a ground triple. The second premise enumerates stored types T of t. The third premise requires that T is the most specific type of all possible types S of t. In other words, there is no type S of t that is a subtype of T. Hence, T is the glb type of the stored types of t.

The implementation view of the above rule is as follows. The schema triples are obtained from the inherited values of the predicates rdfs:domain and rdfs:range. The inherited values have to be the closest when traveling from property p towards the more general properties.

The glb types are now gathered in a $\vee$-type. Hence, the resulting $\vee$-type includes all glb types of t.

$$\frac{\forall T_i \in \mathcal{T}_t,\ t :_{\Downarrow} T_i}{t :_{\Downarrow} \vee[T_i]} \tag{28}$$

The premise says that we identify all triple types T_i that are the individual (glb) $\sqcap$-types of t.

– What is the reason that we have multiple glb types?

Multiple different stored types of t are possible only in the case of multiple inheritance, in the case of the definition of the disjunctive domain/range types, or if predicate is defined for semantically different concepts.
– Describe each possibility in more detail.

4.4 Typing a triple

– Why using $\wedge$ and $\sqcap$ types for typing a triple t?
  – We would like to check typing of a triple $t \in \mathcal{T}_i$.
  – We compute first the ground type $T_g = \wedge[T_1..T_m]$ and a stored type T_s of t.
  – The ground type T_g is computed from the ground types of t's components.
  – The subtype relation should hold $T_g \preceq T_s$.

– Two ways of defining semantics.
– 1) enumeration style: stored types are enumerated as alternatives ($\vee$).
– 2) packed together: alternative types are packed in one $\bigvee$ type.
– One advantage of (1) is that individual glb types can be processed further individually.
– Advantage of (2) is the higer-level semantics without going in implementation.

– Stored types have to be related to all join ground types to represent the correct type of a triple.
– It seems it would be easier to check the pairs one-by-one using (1) in algorithms.
– In case of using complete types in the phases, types would further have to be processed by $\wedge, \vee$ rules.

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., $T \preceq S$.

$$\frac{t \in \mathcal{T}_i \quad T \in \mathcal{T}_t,\, t :_{\downarrow} T \quad S \in \mathcal{T}_t,\, t :_{\uparrow} S \quad T \preceq S}{t : S} \tag{29}$$

– How to compute $T \preceq S$? Refer to position where we have a description.
– Order the possible derivations, gatherings (groupings) ... of types.

– Possible diagnoses.
– Components not related to a top type of a triple?
– Components related to sub-types of a top type?
– Above pertain to all components.

4.5 Typing a graph

– What is a type of a graph?
  – A type of a graph is a graph!
  – It includes a set of triple types forming a schema graph.

– Typing schema triples?
  – What can be checked?
  – Is a schema triple properly related to the super-classes and types of components.
  – Consistency of the placement of a class in an ontology.
  – A class or predicate component not related to other classes?
  – A class or predicate component attached to "conflicting" set of classes? ?
  – Any other examples?

– Typing a graph.
  – Checking whether triple types match in the meeting points.
  – What is the type in meeting points of two triple types?
  – Since a tye of a graph should present any legal triple in $\mathcal{D}$
  –

5 Implementation of type-checking

5.1 Computing LUB ground types

– A ground type of a triple $t = (s, p, o)$ is a type (s_t, p, o_t) such that $(s, \mathit{rdf{:}type}, s_t)$ and $(o, \mathit{rdf{:}type}, o_t)$.
– The sets of types s_t and o_t are stored in the sets g_s and g_o, respectively.
– The ground type of t is then $T_t = (\wedge g_s, p, \wedge g_o)$.
– The lub of a type T_t is computed as follows.

– For each $s_t \in g_s$ ($o_t \in g_o$) compute a closure of a set $\{s_t\}$ ($\{o_t\}$) with respect to the relationship rdfs:superClassOf obtaining the sets c_s and c_o.

– *Each step of the closure newly obtained classes are marked with the number of steps if new in the set, and the maximum of both numbers of steps otherwise.*
– *The maximum guaranties the monotonicity:* $s_t \prec_\downarrow s'_t \Rightarrow m(s_t) > m(s'_t)$.
– *Proof: Assume* $m(s_t) <= m(s'_t)$. *Since* $s_t \prec_\downarrow s'_t$ *then* $m(s'_t) + 1$ *is maximum of* s_t. *Contradiction.*

– *The lub of* T_t *is computed by intersecting all sets* c_s (c_o) *obtained for each* s_t (o_t), *yielding a lub type of* g_s (g_o).
– *The intersection of sets is computed step-by-step. Initially, intersection is the first set* c_s (c_o) *of some* s_t (o_t).
– *In each step, a new set* c_s (c_o) *for another* s_t (o_t) *is intersected with previous result.*
– *The classes that are in the result (intersection) are merged by selecting the maximum number of closure steps for each class in the intersection.*
– *The reason for this is to obtain the number of steps within which all ground classes reach a given lub class.*

– *Finally, the lub classes are selected from the resulted intersection of all* c_s (c_o).
– *The class with the smallest number of steps is taken. Then, it is deleted from the set together with all its super-classes.*
– *If set is not empty the previous step is repeated.*

5.2 Computing GLB stored types

– *We have a tuple* $t = (s, p, o)$.
– *The task is to compute glb of* t*'s stored types.*
– *The algorithm is using the predicates* $p' \preceq p$ *to obtain all domains and ranges of* p.

– *FUNCTION typeGlbStored (p: Property,* cnt*: Integer,* d_p,r_p*: Set): Type*
– *BEGIN*
– *if* $d_p = \{\}$ *then* $d_p = \{c_s \mid (p, \textit{rdfs:Domain}, c_s) \in G\}$
– *if* $r_p = \{\}$ *then* $r_p = \{c_o \mid (p, \textit{rdfs:Range}, c_o) \in G\}$
– *if* $d_p \neq \{\} \wedge r_p \neq \{\}$ *then*
– *RETURN* $(\vee d_p, p, \vee r_p)$
– $ts = \{\}$
– *for* p' $((p, \textit{rdfs:SubPropertyOf}, p') \in G)$
– *BEGIN*
– $T_{p'} = typeGlbStored(p', cnt + 1, d_p, r_p)$
– $ts = ts \cup \{T_{p'}\}$
– *END*
– *RETURN* $\vee ts$
– *END*

5.3 Relating LUB ground and GLB stored types

6 Related work

– Comparing typing relation in an OO model with a KG [10].
– Include the differences between Pierce's (classical) sub-typing view of stored sub-class relationships among classes and the approach taken in this paper
– Pierce treats classes as generators of objects that inherit methods and data members from its super-class.
– The methods are inherited by copying the definitions in each subclass and then explicitely calling the method in the superclass.
– List the differences: classes are identifiers, there is a sub-class relationship included in a subtyping relationship.

7 Conclusions

References

1. V. Bono and M. Dezani-Ciancaglini. A tale of intersection types. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, page 7–20, New York, NY, USA, 2020. Association for Computing Machinery.
2. R. J. Brachman and H. J. Levesque. Knowledge representation and reasoning. Elsevier, 2004.
3. J. Dunfield and N. Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021.
4. L. Ehrlinger and W. Wöß. Towards a definition of knowledge graphs. In *SEMANTiCS*, 2016.
5. J. R. Hindley. *Basic simple type theory*. Cambridge University Press, USA, 1997.
6. 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.
7. A. Hogan, E. Blomqvist, M. Cochez, C. D'amato, G. D. Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, A.-C. N. Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, and A. Zimmermann. Knowledge graphs. *ACM Comput. Surv.*, 54(4), jul 2021.
8. B. C. Pierce. Programming with intersection types, union types, and polymorphism, 1991.
9. B. C. Pierce. Intersection types and bounded polymorphism, 1996.
10. B. C. Pierce. *Types and Programming Languages*. MIT Press, 1 edition, Feb. 2002.
11. Resource description framework (rdf). http://www.w3.org/RDF/, 2004.
12. Rdf schema. http://www.w3.org/TR/rdf-schema/, 2004.
13. I. Savnik. Type-checking graph patterns. Technical Report Technical Report (In preparation), FAMNIT, University of Primorska, 2025.
14. I. Savnik, K. Nitta, R. Skrekovski, and N. Augsten. Type-based computation of knowledge graph statistics. *Annals of Mathematics and Artificial Intelligence*, 2025.