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.

– What is structural part of KG?
– KG domain is complex because of the rich modelling constructs of a KR language.
– The static structure of a KG is built from the identifiers that are bound into triples.
– The identifiers and triples represent the basic structures of a KG, i.e., the values stored in a KG.
– They are the values of languages defined on top of KGs, such as the basic graph-patterns, SparQL queries, and if-then rules.
– 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.
– A KG includes triples that represent the values and triples that represent user-defined types that form the schema of a KG [15].
– Denotational view of classes and type triples.
– The interpretations of classes and triple types form a poset based on the subset relation.

– Facts about the stored typings of the triples.
– In KGs we have types of individual objects represented as classes.
– Further, the types of the triples are the triples including types of triple components.
– A user-defined type of a triple is not linked directly to a triple.
– Types of triples are in a KG defined by specifying the types of identifiers that form triples.
– However, stored types of identifiers do not need to be those that appear in user-defined stored triple types.
– Hence, the types of ground triples must be derived from a KG by selecting the appropriate user-defined triple types from the types of components.

– What types are used for in a KG?
– Types can be used to verify the correctness of the ground triples and the structures that they form.
– The typing errors can appear in a KG if types of identifiers are specified incorrectly.
– 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.
– Typing the triple patterns is similar to typing ground triples [15].
– Before we can define typing of languages that work with a KG, we have to be able to type a triple.

– What is type checking? How it works...
– Types represent a higher-level description of ground triples.
In type theory, types represent concepts that are used to classify the values from a given

language [11, 5]. Correct typing assures that the functions are applied to correct parameters in a program. The type-checking problem

– Having a program verify that it conforms with the typing rules.

– In this process the rules can be applied in two directions.

– Type-assignment derives a type to an expression in a bottom-up manner.

– Here we use typing rules in the forward direction, hance type inference.

– Verifying that an expression adheres a given type uses typing rules in a backward direction.

– Here the rules decompose expressions into syntactic components and verifies recursively the types of comonents.

– In order to check the types of a KG, the type of each ground triple has to be checked.
– There are two ways of computing the type of a triple.
– First, we can use types of identifiers representing the components of a triple.

– We call this type a ground type *since the types of identifiers classes from the bottom of the class ontology.*

– Note that, because of the rule of subsumption [11], the type of a ground triple is any triple type that is a supertype of a ground triple type.
– Second, we can select appropriate user-defined triple type T such that the components of T are the types of the components of the ground triple we are checking.

– The components of the user-defined triple types are noramlly from the top of the class ontology.

– Type checking of a ground triple is converted to checking the relationships between the ground type and user-defined type of a triple.
– In all contexts we want to determine minimal type *of a triple, having the smallest interpretation.*

2 Preliminaries

2.1 Knowledge graph

This section defines a knowledge graph as a RDF graph [12] using RDF-Schema [13] 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 [1] 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 [11, 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 [12, 13], 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 [11, 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 5. 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 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 [13]. 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.

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

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 4.

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

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

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

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

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* [11].

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

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, $\wedge[T_i^{i\in[1,n]}]$ is the greatest lower bound (abbr. GLB) of types $T_i^{i\in[1,n]}$ [9, 10]. 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 $\tau = \mathcal{I}_c \cup \mathcal{T}_t$ is used to ground the types in the rules.

The instances of a type $\wedge[T_i^{i\in[1,n]}]$ are the instances of all particular types T_i. This is stated by the following rule.

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

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 $\wedge[T_i^{i\in[1,n]}]$.

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

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, $\vee[T_i^{i\in[1,n]}]$ is the least upper bound of types $T_i^{i\in[1,n]}$ [9].

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

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

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

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 $\forall i \in [1..n],\ T_i \in \tau$.

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

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

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

Note that the interpretation of a class C is a set of instances $[\![C]\!]_\Delta = \{I \mid I \in \mathcal{I}_i \wedge I : C\}$. Further, the interpretation of a triple type T is the set of ground triples $[\![T]\!]_\Delta = \{t \mid t \in \mathcal{T}_t \wedge t : T\}$[1][15].
– What about subtyping?

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, and .

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 [11], respectively. However, in a KG we are also interested in the upper bound and lower bound types [2]. 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) [8]. 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 $T_i^{i\in[1,n]}$], $M = S \sqcap [T_i]$, 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 $M_j \preceq T_i^{i\in[1,n]}$, 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.

[1] Triple types $\mathcal{T}_t$ are presented in the following Section 5.

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 14 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* 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.

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 $\wedge[T_i^{i\in[1,n]}]$ 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 solely the interpretation of M. The set $\bigcap[\![T_i]\!]_\Delta$ can also include objects that are not instances of M. Hence,

$$[\![\sqcap[T_i^{i\in[1,n]}]]\!]_\Delta = [\![M]\!]_\Delta \subseteq [\![\wedge[T_i^{i\in[1,n]}]]\!]_\Delta = \bigcap_{i\in[1..n]} [\![T_i]\!]_\Delta.$$

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 [14]. 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 graph pattern variable.

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

- *The use of $\wedge$ and $\vee$ types to describe identifiers.*
  - *$\upsilon = \mathcal{I}_i \cup \mathcal{T}_i$ and $\tau = \mathcal{I}_c \cup \mathcal{T}_t$*
  - *General rules are defined to work with identifiers and triples.*
  - *Hence typing rules can be used to type idents and triples.*

- *For $V \in \upsilon$ gather ground types of identifiers with $\wedge$-type as $V :_\downarrow \wedge[T_i^{i\in[1,n]}]$.*
  - *$V \in \upsilon \quad \forall T_i \in \tau,\ t :_\downarrow T_i$.*
  - *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 $V \in \upsilon$.*

$$\frac{V \in \upsilon \quad T_i^{i\in[1,n]} \in \tau \quad V :_{\downarrow} T_i^{i\in[1,n]}}{V :_{\downarrow} \wedge[T_i^{i\in[1,n]}]} \tag{16}$$

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 [6], 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_s have to include subclasses of C_g.

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 set $\{T_i^{i\in[1,n]}\}$ with respect to $\preceq$. The super-types of the minimal elements of $\{T_i^{i\in[1,n]}\}$ include all valid types of V. 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 type and the user-defined type of V.

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.

$$\frac{V \in \upsilon \quad V :_{\downarrow} \wedge[T_i^{i\in[1,n]}] \quad S \in \{T_k^{k\in[1,n]}\} \quad \forall i \in [1,n],\ S \preceq T_i \vee S \not\sim T_i}{V :_{\Downarrow} S} \tag{17}$$

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 $\wedge[S_1..S_m]$. The interpretation of a minimal ground type $[\![\wedge[S_1..S_m]]\!]$ of V is minimal since $\wedge[S_1..S_m]$ includes a set of minimal and unrelated types of V. Therefore, $[\![\wedge[S_1..S_m]]\!] \subseteq [\![\wedge[T_i^{i\in[1,n]}]]\!]$.

$$\frac{V \in \upsilon \quad \forall i \in [1,m],\ V :_{\Downarrow} S_i}{V :_{\Downarrow} \wedge[S_j^{j\in[1,m]}]} \tag{18}$$

The rule for filtering $\wedge[T_i^{i\in[1,n]}]$ of all $T_i \succeq T_j$ where $i \neq j$ by using algorithmic typing is defined as follows. The algorithm implementing the operation MIN is presented in Section 6.1.

$$\frac{V \in \upsilon \quad \mapsto V :_{\downarrow} \wedge[T_i^{i\in[1,n]}] \quad \mapsto \{S_j^{j\in[1,m}\} = \Downarrow[T_i^{i\in[1,n]}]}{\mapsto V :_{\Downarrow} \wedge[S_j^{j\in[1,m]}]} \tag{19}$$

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 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.

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

Let us now present the typing rules that, given $V \in \upsilon$, determine a join type of $T_i^{i\in[1,n]}$ as $V : \sqcup[T_i^{i\in[1,n]}]$. A join type $\sqcup[T_i^{i\in[1,n]}]$ is defined on the set of types ordered into a poset. Since the types form a poset, we may have more than one join types. A join type $\sqcup[T_i^{i\in[1,n]}]$ is a set of minimal types $S_j^{j\in[1,m]}$ that are related to all types $T_i^{i\in[1,n]}$ via $\preceq$. Precisely, a join type is a set of the least upper bound types of $\{T_i^{i\in[1,n]}\}$ with respect to the relation $\preceq$.

Rules 20-21 present the logical definition of the join type. The following Rule 20 determines one join type but can be used to generate all join types.

$$\frac{V \in \upsilon,\, V :_{\downarrow} \wedge[T_i^{i\in[1,n]}] \quad S \in \tau,\, T_i^{i\in[1,n]} \preceq S \quad \forall P \in \tau,\, (T_i^{i\in[1,n]} \preceq P \wedge S \preceq P) \vee P \not\prec S}{V :_{\sqcup} S} \tag{20}$$

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

$$\frac{V \in \upsilon \quad S_i^{i\in[1,m]} \in \tau \quad \forall i \in [1,m],\, V :_{\sqcup} S_i}{V :_{\sqcup} \wedge[S_i^{i\in[1,m]}]} \tag{21}$$

The following Rule 22 derives the complete join type in one step. The operator $\sqcup[T_i^{i\in[1,n]}]$ returns as a result a set of LUB types of $T_i^{i\in[1,n]}$. Since all the join types are valid types of V, we can group them into one $\wedge$ type.

$$\frac{V \in \upsilon \quad \mapsto V :_{\downarrow} \wedge[T_i^{i\in[1,n]}] \quad \mapsto \{S_j^{j\in[1,m]}\} = \sqcup[T_i^{i\in[1,n]}]}{\mapsto V :_{\sqcup} \wedge[S_j^{j\in[1,m]}]} \tag{22}$$

- *Integrate this explanation above.*
- *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 V are used to compute a type of V.*
  - *$V :_{\sqcup} S$ 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.*

- *Example from a KG.*
- *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.*

5 Typing triples

– *We would like to type of a triple $t \in \mathcal{T}_i$.*
– *There are two basic aspects of a triple type.*
  – *1. $t : T_g$ is computed bottom-up: from the stored types of triple components.*
  – *2. $t : T_u$ can be computed from the user-defined types of properties.*
  – *The relation $T_g \preceq T_u$ must hold if the typing of KG is correct.*
  – *If the predicate p of type T_u is defined in multiple contexts, some of disjunctively*
  – *linked components of T_u may not be related to T_g.*
  – *The filtering of T_u is done by Rule 33.*

– *About the types that are computed bottom-up.*
  – *Ground type of a triple is computed first by extending $:_\downarrow$ to triples.*
  – *A triple can have multiple ground types $T_g = (\wedge[S_i^{i\in[1,k]}], p, \wedge[O_j^{j\in[1,m]}])$.*
  – *Next, the join type $\sqcup[T_g]$ of ground type T_g is derived.*
  – *The type $\sqcup[T_g]$ is used as a stepping stone to determine the final type of t.*
    – *To be used for type-checking graph patterns.*

– *Stored triple types are user-defined types.*
  – *Stored types for a predicate p are defined via the predicates rdfs:domain and rdfs:range.*
  – *From the top of the ontology, the stored type $:_\uparrow$ is determined based on p.*
  – *However, given p we can have a triple type (T_s, p, T_o) such that T_s or T_o are defined for some $p' \succeq p$.*
  – *Special case: p' has two domains T_s^1 and T_s^2—type is then $(T_s^1 \vee T_s^2, p, \_) \equiv (T_s^1, p, \_) \vee (T_s^2, p, \_)$.*
  – $\mathcal{T}_\uparrow = \{(T_s, p, T_o) | p \in \mathcal{I}_p \wedge (p, \mathit{rdfs{:}domain}, T_s) \in \Delta \wedge (p, \mathit{rdfs{:}range}, T_o) \in \Delta\}$
  – *Derived types of the stored types are computed using Rule 11.*
  – *Derived types of $\mathcal{T}_\uparrow$ include the complete top of the ontology*

– *Stored triple types for a given predicate p are computed as MIN of valid stored types for p.*
  – *The MIN types of types obtained using $:_\uparrow$ are the smallest triple types*
  – *including MIN classes as components.*
  – *Stored type have to be minimal to have minimal interpretaton (e.g., type of a triple pattern).*
  – *Finally, the type $:$ of t is deterimned by summing alternative $:_\Downarrow$ types.*
  – *Note there can be more than one $\Downarrow$-type.*
  – *This happens when triple types include property that is defined in two different contexts.*

– *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: *Discussion on special role of predicates and their relations to classes?*
  – *Mention Cyc as the practical KB with rich hierarchy of predicates.*

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 type (least upper bound). The stored ground type includes types that are stored in a KG. The minimal ground type then consists of solely the minimal gound types. Finally, the join type is a least upper bound type of the minimal ground type.

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, 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 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.

$$\frac{t \in \mathcal{T}_i,\ t = (I_s, p, I_o) \quad I_s :_\downarrow T_s \quad I_o :_\downarrow T_o \quad p :_\downarrow \text{rdf:Property}}{t :_\downarrow T_s * p * T_o} \tag{23}$$

The type T_s is either a class identifier or a $\wedge$-type composed of a conjunction of minimal set of class identifiers. 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.

A triple can have multiple ground types. 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 23.

$$\frac{\forall i \in [1..n],\ t :_\downarrow T_i}{t :_\downarrow \wedge[T_i]} \tag{24}$$

– *Typing using lub types of* $T_i^{i\in[1,n]}$. *Explain why this is needed?*

Let us now define the least upper bound types (abbr. *lub*) of ground types derived by Rule 24. 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 $T_i^{i\in[1,n]}$ are computed in two steps as before when gathering multiple ground types with conjunction. A single lub type is defined as follows.

$$\frac{t :_\downarrow \wedge[T_i^{i\in[1,n]}] \quad T \in \mathcal{T}_t, \quad \forall i,\ T_i \preceq T \quad \forall S \in \mathcal{T}_t,\ \forall i,\ (T_i \preceq S \wedge T \preceq S) \vee T \not\succ S}{t :_\sqcup T} \tag{25}$$

$$\frac{t :_{\downarrow} \wedge[T_i^{i\in[1,n]}] \quad \mapsto T = \sqcup[T_i^{i\in[1,n]}]}{t :_{\downarrow\sqcup} T} \tag{26}$$

$$\frac{t :_{\downarrow} \wedge[T_i^{i\in[1,n]}] \quad \mapsto \forall i \in [1..m],\ S_i = \sqcup[T_i^{i\in[1,n]}]}{t :_{\downarrow\sqcup} \wedge[S_1..S_m]} \tag{27}$$

The above rule states that a type T is a lub type of a ground type $\wedge[T_i^{i\in[1,n]}]$ 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.

$$\frac{\forall i \in [1..n](t :_{\sqcup} T_i)}{t :_{\sqcup} \wedge[T_i^{i\in[1,n]}]} \tag{28}$$

5.2 Stored types of triples

– *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 type is constructed by selecting types including a predicates $p' \succeq p$ as the domains and ranges.

$$\frac{t \in \mathcal{T}_i,\ t = (s, p, o) \quad p' \in \mathcal{I}_p, p \preceq p' \quad (p', \text{domain}, T_s) \in g \quad (p', \text{range}, T_o) \in g}{t :_{\uparrow} T_s * p * T_o} \tag{29}$$

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.

$$\frac{t \in \mathcal{T}_i, t = (s, p, o) \quad p_1, p_2 \in \mathcal{I}_p \quad p \preceq p_1 \quad p \preceq p_2 \quad (p_1, \text{domain}, T_s) \in g \quad (p_2, \text{range}, T_o) \in g}{t :_{\uparrow} T_s * p * T_o} \tag{30}$$

– *If p inherits from multiple $p' \succeq p$, then the above rule generates multiple types. Explain.*

– *The type is determined only if the domain and range of $p' \succeq p$ 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 29.

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

$$\frac{t \in \mathcal{T}_i \quad T \in \mathcal{T}_t,\, t :_{\uparrow} T \quad \forall S \in \mathcal{T}_t,\, t :_{\uparrow} S \quad T \preceq S \vee T \not\sim S}{t :_{\Downarrow} T} \tag{31}$$

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 MIN type of the stored types of t.

- *What is the meaning of triple types that are* not related *(*$\not\sim$*).*
  - *1. This can be either that we have two* p *roots with unrelated MIN triple types.*
  - *This is possible only if* p *is defined for semantically different concepts.*
  - *2. Two* p*-rooted but unrelated stored types through multiple inheritance.*
  - *Therefore, we can have more than one stored MIN types.*

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

$$\frac{\forall T_i \in \mathcal{T}_t,\, t :_{\Downarrow} T_i}{t :_{\Downarrow} \vee[T_i]} \tag{32}$$

The premise says that we identify all triple types T_i that are the MIN types $t :_{\Downarrow} T_i$. The MIN types are gathered in a triple type $\vee[T_i]$.

5.3 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* $T_g = \wedge[T_i^{i\in[1,n]}]$ *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., $T \preceq S$.

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

– 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.

6 Implementation of type-checking

6.1 Computing MIN type

– We have a tuple $t = (s, p, o)$.
  – The task is to compute MIN type 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

– Start with a set $\{p\}$ and close the set by using rdfs:superPropertyOf marking them with the "distance" from p.

– Gather all domain and range types of $p' \preceq p$ in d_p and r_p, respectively, marking each domain and range class with the distance of the related p'.

– Generate types $\vee(s_t, p, o_t)$ where $s_t \in d_p$, $o_t \in r_p$ and both s_t and o_t are marked with the minimal values.

– Note that there can be more than one element s_t (and o_t) marked with a minimal value in d_p (and r_p). Hence Cartesian product of the selected domains and ranges are used to generate types (s_t, p, o_t).

6.2 Computing join type

– 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.

6.3 Relating LUB ground and MIN stored types

7 Related work

– Comparing typing relation in an OO model with a KG [11].
– The values from KGs have similar structure to the values of object-oriented models.
– However, the predicates of a KG are more expressive than the data members of the classes.
– Similarly, the record types form a lattice under subtype relationship with least upper bound and greatest lower bound based on sets of record attributes.

– 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.

8 Conclusions

References

1. R. J. Brachman and H. J. Levesque. *Knowledge Representation and Reasoning*. Elsevier, 2004.
2. B. A. Davey and H. A. Priestley. *Introduction to lattices and order*. Cambridge University Press, 2nd edition, 2002.
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. S. B. Knudstorp. Tho modal logic of minimal upper bounds, 2024.
9. B. C. Pierce. Programming with intersection types, union types, and polymorphism, 1991.
10. B. C. Pierce. Intersection types and bounded polymorphism, 1996.
11. B. C. Pierce. *Types and Programming Languages*. MIT Press, 1 edition, Feb. 2002.
12. Resource description framework (rdf). http://www.w3.org/RDF/, 2004.
13. Rdf schema. http://www.w3.org/TR/rdf-schema/, 2004.
14. I. Savnik. Type-checking graph patterns. Technical Report Technical Report (In preparation), FAMNIT, University of Primorska, 2025.
15. I. Savnik, K. Nitta, R. Skrekovski, and N. Augsten. Type-based computation of knowledge graph statistics. *Annals of Mathematics and Artificial Intelligence*, 2025.