

# Type-checking knowledge graphs

Iztok Savnik<sup>1</sup>

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

**Abstract.** We first present a formal view of a knowledge graph. On this basis, the type-checking rules are developed to define correct typing relationships among the triples of a knowledge graph. We discuss the algorithms for verifying the typing relationships against the given knowledge graph. Finally, we present the experimental results of type-checking the Yago4 knowledge graph.

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

## Table of Contents

Type-checking knowledge graphs .....	1
<i>Iztok Savnik</i>	
1 Introduction .....	3
2 Preliminaries .....	4
2.1 Knowledge graph .....	4
2.2 Rule language .....	4
3 Typing identifiers .....	5
3.1 Typing literals .....	5
3.2 Intersection and union types .....	5
3.3 Stored typing and subtyping of identifiers .....	6
3.4 Typing and subtyping identifiers .....	7
4 Typing triples .....	8
4.1 Product types .....	8
4.2 Deriving a ground type of a triple .....	8
4.3 Stored types of triples .....	9
4.4 Typing a triple .....	11
4.5 Typing a graph .....	12
Typing schema triples .....	12
5 Implementation of type-checking .....	12
5.1 Computing LUB ground types .....	12
5.2 Computing GLB stored types .....	13
5.3 Relating LUB ground and GLB stored types .....	14
6 Conclusions .....	14

## 1 Introduction

- Introduction to knowledge graphs (KG)... [5, 3].
- 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... [8, 4].
- 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 [8, 1, 6].
  
- 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 property 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 ids 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 [2] 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 [9] using RDF-Schema [10] 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  $O = I \cup B \cup L$ .

Let  $I$  be the set of URI-s,  $B$  the set of blanks and  $L$  be the set of literals. Let us also define sets  $S = I \cup B$ ,  $P = I$ , and  $O = I \cup B \cup L$ .

*RDF triple* is a triple  $(s, p, o) \in S \times P \times O$ . *RDF graph*  $g \subseteq S \times P \times O$  is a set of triples. Set of all graphs will be denoted as  $G$ . We suppose the existence of a set of variables  $V$  and the set of *terms*  $T = O \cup V$ . Term  $t \in T$  is ground if  $t \in O$ .

We say that RDF graph  $g_1$  is *sub-graph* of  $g_2$ , denoted  $g_1 \sqsubseteq g_2$ , if all triples in  $g_1$  are also triples from  $g_2$ .

– Define major structure of KG on the basis of the sorts of data.

– ...the set  $I$  includes individual identifiers  $I_i$ , class identifiers  $I_c$  and predicate identifiers  $I_p$ .

### 2.2 Rule language

– Put somewhere general discussion on rules.

– What elements of rule are reasonable. Criterium: tracable elements of rule.

– What is tracable? Looping on a "selected" set. Loop may go through elements of a set, all more specific/general elements, etc.

– They are extended to handle poset of identifiers that can form triples.

– We do not have variables with types assigned in  $\Gamma$  but facts that are stored in a KG.

– Things stored in a KG are classified as identifiers and triples, and then further into individual and type objects ( $\mathcal{I}$  and  $\mathcal{T}$ ).

– In this way we obtain individual identifiers  $\mathcal{I}_i$  and class identifiers  $\mathcal{I}_c$ , as well as the individual triples  $\mathcal{T}_i$  and triple types  $\mathcal{T}_t$ .

– We specify the domains (sets) of objects (identifiers, triples, classes and types).

– 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  $S$ .

– Interpretation of the rules.

– Forward: generative view of rules where rules enumerate the possible conclusions from the premises that generate objects from their domains. We are generating possible results.

– Backward: for a selected object from a domain we check the construction by means of rule(s).

### 3 Typing identifiers

- *Intro to typing idents.*
- *Typing idents without considering info about the triples.*

The individual and class entities are represented by identifiers from the set  $\mathcal{I}$ . The individual identifiers  $\mathcal{I}_i$  stand for literals, concrete and abstract entities. The class identifiers  $\mathcal{I}_c$  represent abstract entities that have an unempty interpretation. The abstract entities include, besides the identifiers of user-defined classes, the types (classes) of literals.

A graph database includes stored definitions for typing the individual identifiers, and for representing the specialization/generalization hierarchies of classes and properties.

- *Details.*
- *1. First define base type of identifiers  $:_1$  and stored subtyping relationship  $\preceq_1$ .*
- *2. From the basis define the indent typing  $:$  and subtyping rel  $\preceq$  among identifiers.*
- *3. Include the link between subtyping and typing.*
- *4. Define lub type using  $\wedge$  type for a given ground ident.*

#### 3.1 Typing literals

Literals are the values of an atomic type. The atomic types are in RDF provided by the RDF-Schema dictionary [10]. 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.

$$\frac{L \in \mathcal{I}_i, T \in \mathcal{I}_c, "L"^\wedge T \in \mathcal{L}}{L : T} \quad (1)$$

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

#### 3.2 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,  $\wedge[T_1 \dots T_n]$  is the greatest lower bound of types  $T_1 \dots T_n$  [6, 7].

$$T_1 \wedge T_2 \preceq T_1 \quad (2)$$

$$T_1 \wedge T_2 \preceq T_2 \quad (3)$$

$$\wedge[T_1 \dots T_n] \preceq T_i \quad (4)$$

If the type  $S$  is more specific than the types  $T_1 \dots T_n$  then  $S$  is more specific than  $\wedge[T_1 \dots T_n]$ . First, we present the rule for a pair of types  $T_1$  and  $T_2$ .

$$\frac{S \preceq T_1 \quad S \preceq T_2}{S \preceq T_1 \wedge T_2} \quad (5)$$

$$\frac{\forall i, S \preceq T_i}{S \preceq \wedge[T_1 \dots T_n]} \quad (6)$$

The intersection and union types are dual. This can be seen also from the rules that are used for each particular type.

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. The type  $T_1 \vee T_2$  is the smallest upper bound of the types  $T_1$  and  $T_2$ . In general,  $\vee[T_1 \dots T_n]$  is the smallest upper bound of types  $T_1 \dots T_n$  [6].

$$T_1 \preceq T_1 \vee T_2 \quad (7)$$

$$T_2 \preceq T_1 \vee T_2 \quad (8)$$

$$T_i \preceq \vee[T_1 \dots T_n] \quad (9)$$

If the type  $T$  is more general than the types  $S_1 \dots S_n$  then  $T$  is more general than  $\vee[S_1 \dots S_n]$ . First, we present the rule for types  $T_1$  and  $T_2$ .

$$\frac{S_1 \preceq T \quad S_2 \preceq T}{S_1 \vee S_2 \preceq T} \quad (10)$$

$$\frac{\forall i, S_i \preceq T}{\vee[S_1 \dots S_n] \preceq T} \quad (11)$$

### 3.3 Stored typing and subtyping of identifiers

Let us introduce the typing and specialization/generalization relationships formally. 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 sub-class relationship between the class identifiers  $i_1$  and  $i_2$ . The index ' $\downarrow$ ' in relations  $:_{\downarrow}$  and  $\preceq_{\downarrow}$  denotes that the relationships is stored in a database. Such notation allows us to address differently the stored and the derived parts of the graph database schema.

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

$$\frac{I \in \mathcal{I}_i \quad I_c \in \mathcal{I}_c \quad (I, \text{rdf:type}, I_c) \in \mathcal{D}}{I :_{\downarrow} I_c} \quad (12)$$

The individual entity  $I$  can have more than one stored types, e.g.,  $I_{c1}$  and  $I_{c2}$ . Therefore,  $I :_{\downarrow} I_{c1}$  and  $I :_{\downarrow} I_{c2}$  holds, and we can instead write  $I :_{\downarrow} I_{c1} \wedge I_{c2}$ . All existing stored typings of  $I$  can be gathered by Rule 23 presented later.

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

$$\frac{I_1, I_2 \in \mathcal{I}_c \quad (I_1, \text{rdfs:subClassOf}, I_2) \in \mathcal{D}}{I_1 \preceq_{\downarrow} I_2} \quad (13)$$

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

$$\frac{I_1, I_2 \in \mathcal{I}_p \quad (I_1, \text{rdfs:subPropertyOf}, I_2) \in \mathcal{D}}{I_1 \preceq_{\downarrow} I_2} \quad (14)$$

### 3.4 Typing and subtyping identifiers

The one-step relationships  $\preceq_{\downarrow}$  and  $\preceq$  are now extended with the reflexivity and transitivity to obtain the relationships  $\preceq$  and  $\preceq$ . The relation  $\preceq$  forms a partial ordering of class identifiers.

First, the one-step relationship  $\preceq_s$  is generalized to the relationship  $\preceq$  defined over class identifiers  $\mathcal{I}_c$ .

$$\frac{I_1, I_2 \in \mathcal{I}_c \quad I_1 \preceq_{\downarrow} I_2}{I_1 \preceq I_2} \quad (15)$$

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

$$\frac{S \in \mathcal{I}_c}{S \preceq S} \quad (16)$$

The subtype relationship is also transitive.

$$\frac{S, U, T \in \mathcal{I}_c \quad S \preceq U \quad U \preceq T}{S \preceq T} \quad (17)$$

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

$$\frac{S, U \in \mathcal{I}_c \quad S \preceq U \quad U \preceq S}{S = T} \quad (18)$$

As a consequence of the rules 16-18 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`. The following rule specifies that all class identifiers are more specific than  $\top$ .

$$\frac{\forall S \in \mathcal{I}_c}{S \preceq \top} \quad (19)$$

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

$$\frac{I \in \mathcal{I}_i \quad C \in \mathcal{I}_c \quad I \preceq_{\downarrow} C}{I \preceq C} \quad (20)$$

The link between the typing relation and subtype relation is provided by adding a typing rule called *rule of subsumption* [8].

$$\frac{I \in \mathcal{I}_i \quad S \in \mathcal{I}_c \quad I : S \quad S \preceq T}{I : T} \quad (21)$$

- Properties have dual role: they are instances and types at the same time.
- Present the features of properties from this point of view.

## 4 Typing triples

- 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 determined by summing alternative  $:\uparrow$  types.
  
- Unfinished!!
- 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?
  
- Unfinished!!
  
- 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  $t = (s, p, o)$  is a triple  $T = T_s * p * T_o$  that includes the ground types of  $t$ 's components  $s$  and  $o$ , and the property  $p$  which now has the role of a type. A ground type of a triple is defined by the following rule.

$$\frac{t \in \mathcal{T}_i, t = (s, p, o) \quad I_s, I_o \in \mathcal{I}_c, s :_{\downarrow} I_s \wedge o :_{\downarrow} I_o \quad p :_{\downarrow} \text{rdf:Property}}{t :_{\downarrow} I_s * p * I_o} \quad (22)$$

The class  $I_s$  is one of the ground types of  $s$ , and the type  $I_o$  is one of the ground types of  $o$ . 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  $T_1, \dots, T_n$  are obtained using Rule 22.

$$\frac{t \in \mathcal{T}_i \quad \forall T_i \in \mathcal{T}_t, t :_{\downarrow} T_i}{t :_{\downarrow} \wedge[T_i]} \quad (23)$$

– *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 23. Since a partially 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.

$$\frac{t :_{\downarrow} \wedge[T_1..T_n] \quad T \in \mathcal{T}_t \quad \forall i, T_i \preceq T \quad \forall S \in \mathcal{T}_t, \forall i, T_i \preceq S \wedge T \preceq S}{\vdash t :_{\sqcup} T} \quad (24)$$

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  $T_1, \dots, T_n$ . The lub types can be now gathered using the following rule.

$$\frac{\forall i, T_i \in \mathcal{T}_t \quad t :_{\sqcup} T_i}{t :_{\sqcup} \wedge[T_1..T_n]} \quad (25)$$

### 4.3 Stored types of triples

– *General comments.*

– *Analysis tool. Show minimality of the stored types (either enumerated or gathered with  $\vee$ ).*

– *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.

$$\frac{t \in \mathcal{T}_i, t = (s, p, o) \quad p' \in \mathcal{I}_p, p \preceq p' \quad (p', \text{domain}, T_s) \in g \quad (p', \text{range}, T_o) \in g}{t :_{\uparrow} (T_s, p, T_o)} \quad (26)$$

- Comments and description of the above rule.
- Note  $p$  is used for all types.  $p$  should 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 particularly 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.

$$\frac{t \in \mathcal{T}_i, t = (s, p, o) \quad p_1, p_2 \in \mathcal{I}_p \quad p \preceq p_1 \quad p \preceq p_2 \quad (p_1, \text{domain}, T_s) \in g \quad (p_2, \text{range}, T_o) \in g}{t :_{\uparrow} (T_s, p, T_o)} \quad (27)$$

- 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 should 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 26.

- 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.

$$\frac{t \in \mathcal{T}_i \quad T_i \in \mathcal{T}_t, t :_{\uparrow} T_i \quad \forall S \in \mathcal{T}_t, t :_{\uparrow} S \quad T_i \preceq S \vee T_i \not\sim S}{t :_{\sqcap} T_i} \quad (28)$$

The first premise says that  $t$  is a ground triple. The second premise enumerates stored types  $T_i$  of  $t$ . The third premise requires that  $T_i$  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  $T_i$ . Hence,  $T_i$  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  $t$ .

$$\frac{\forall T_i \in \mathcal{T}_t, t :_{\sqcap} T_i}{t :_{\sqcap} \vee[T_i]} \quad (29)$$

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

- Two ways of defining semantics.
- 1) enumeration style: stored types are enumerated as alternatives ( $\vee$ ).
- 2) packed together: alternative types are packed in one  $\vee$  type.
- One advantage of (1) is that individual glb types can be processed further individually.
- Advantage of (2) is the higher-level semantics without going in implementation.
- Now stored types have to be related to all lub base 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.,  $T \preceq S$ .

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

- 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 schema triples forming a schema graph.
- Typing a graph bottom-up?
- Checking that all the triples are of correct types.

#### 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. What is this?
- A class or predicate component not related to other classes?
- A class or predicate component attached to “conflicting” set of classes? What can be detected?
- Any other examples?

## 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  $(s, \text{rdf:type}, s_t)$  and  $(o, \text{rdf:type}, o_t)$ .
- 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  $m(s_t) \leq m(s'_t)$ . 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  $d_p = \{c_s \mid (p, \text{rdfs:Domain}, c_s) \in G\}$
- if  $r_p = \{\}$  then  $r_p = \{c_o \mid (p, \text{rdfs:Range}, c_o) \in G\}$
- if  $d_p \neq \{\} \wedge r_p \neq \{\}$  then
- RETURN  $(\vee d_p, p, \vee r_p)$
- $ts = \{\}$
- for  $p' ((p, \text{rdfs:SubPropertyOf}, p') \in G)$
- BEGIN
- $T_{p'} = \text{typeGlbStored}(p', cnt + 1, d_p, r_p)$
- $ts = ts \cup \{T_{p'}\}$
- END
- RETURN  $\vee ts$

–END

### 5.3 Relating LUB ground and GLB stored types

## 6 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. J. Dunfield and N. Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021.
3. L. Ehrlinger and W. Wöß. Towards a definition of knowledge graphs. In *SEMANTiCS*, 2016.
4. J. R. Hindley. *Basic simple type theory*. Cambridge University Press, USA, 1997.
5. 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.
6. B. C. Pierce. Programming with intersection types, union types, and polymorphism, 1991.
7. B. C. Pierce. Intersection types and bounded polymorphism, 1996.
8. B. C. Pierce. *Types and Programming Languages*. MIT Press, 1 edition, Feb. 2002.
9. Resource description framework (rdf). <http://www.w3.org/RDF/>, 2004.
10. Rdf schema. <http://www.w3.org/TR/rdf-schema/>, 2004.