

A Formal Analysis of Classes that Act as Individuals in Wikidata: Punning Intended

Margherita Necchi¹, Thomas Bonald¹ and Fabian M. Suchanek¹

¹*Institut Polytechnique de Paris, France*

Abstract

Large-scale knowledge bases like Wikidata and YAGO often include entities that must act as both classes and named individuals – a phenomenon known as *punning*. For example, the Nobel Prize in Physics, as a recurring event, serves as both an individual (with properties such as its founding year) and a class (whose instances are its annual editions). In this work, we conduct an empirical analysis of Wikidata, which shows that punning affects thousands of classes, covering categories such as recurring events, products, proteins, genes, colors, and others. We then formalize the requirements needed for a knowledge representation to support punning entities while preserving existing reasoning capabilities. We compare several formalisms, and find that none of them fulfills all desiderata to represent punning out of the box.

Keywords

Knowledge graphs, punning, Wikidata, YAGO, OWL, ontology design, class-instance duality

1. Introduction

Knowledge representation systems traditionally rely on a fundamental ontological separation between classes and individuals [1, 2]. Classes provide intensional descriptions of entity sets and support taxonomic reasoning through subclass hierarchies, while individuals denote extensional, concrete entities subject to factual assertions.

However, real-world knowledge frequently violates this dichotomy – a phenomenon called *punning*, where a single entity serves as a class in the terminological layer and as an individual in the assertional layer. Wikidata, a large-scale collaborative knowledge base [3], frequently embraces punning and even necessitates it. For instance, the entity *Nobel Prize in Physics* is treated both as an individual (carrying Class facts such as its founding year) and as a class (whose instances are its annual editions). In such cases, neither perspective alone captures the full scope of information editors aim to record.

Wikidata can accommodate punning because it does not prioritize automated reasoning or strict logical consistency. In contrast, YAGO positions itself as a logically consistent subset of Wikidata [4, 5]. This consistency is critical for filtering erroneous statements and enabling logical inference. To achieve this, YAGO must distinguish between existential statements (e.g., “Marie Curie has a child named Irène Curie”) and general statements (e.g., “All children of a person are persons”). The former asserts the existence of a specific individual without imposing constraints on others, while the latter imposes constraints on all potential instances without asserting existence. To make this distinction, standard reasoning formalisms (including OWL and description logics) enforce a strict separation between individuals and classes. YAGO further adheres to standard RDFS/OWL entailment (transitive subclass and type inference, domain/range checking, and class disjointness axioms), which remain well-behaved only if every entity is exclusively a class or an individual. Allowing an entity to be both can trigger incorrect disjointness axioms or create ambiguity in domain/range checks. Thus, strict separation is a prerequisite for YAGO’s logical consistency.

The central challenge, then, is how to represent punning within logical frameworks like YAGO. This

6th Wikidata Workshop, co-located with ISWC 2026, October 2026, Bari, Italy

✉ margherita.necchi@ip-paris.fr (M. Necchi); thomas.bonald@ip-paris.fr (T. Bonald); fabian@suchanek.name (F. M. Suchanek)

🌐 <https://perso.telecom-paris.fr/bonald/> (T. Bonald); <https://suchanek.name> (F. M. Suchanek)

🆔 0009-0002-0011-3788 (M. Necchi); 0000-0003-0468-0384 (T. Bonald); 0000-0001-7189-2796 (F. M. Suchanek)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

issue is not limited to YAGO, as Wikidata itself currently suffers from this ambiguity. For example, users cannot determine whether a P279 (subclass of) edge represents a genuinely functional class or a data-entry artifact with no operational meaning. A principled treatment of punning would also enable tools such as reasoners, consistency checkers, and federated query engines to work better on Wikidata.

In this paper, we first investigate the prevalence of punning in Wikidata. We then distill recurring modeling patterns into a representative use case, formalize the requirements for a robust solution, and evaluate candidate Semantic Web formalisms against them. We find that, although several mechanisms address many desiderata or can be extended to address more of them, none of the formalisms satisfies all desiderata out of the box.

2. Punning in Wikidata at Scale

2.1. Method

Wikidata organises knowledge around items, which have properties. The two most relevant properties for our analysis are P31 (instance of) and P279 (subclass of), corresponding respectively roughly to `rdf:type` and `rdfs:subClassOf`. An item carrying both relations is a punning entity in the Wikidata model. To confirm that punning is a systematic, cross-domain phenomenon and not merely an artefact, we enumerate and analyse every punning item in Wikidata through the following SPARQL query:

```
SELECT DISTINCT ?item WHERE {
  ?item wdt:P31 ?class1 .
  ?item wdt:P279 ?class2 .
}
```

This gives around **4M** distinct punning items (as of September 2026). We then count the distinct classes that punning items are typed against:

```
SELECT DISTINCT ?class1 WHERE {
  ?item wdt:P31 ?class1 .
  ?item wdt:P279 ?class2 .
}
```

This returns around **23k** classes. However, a subset of these are not genuine punning: if the class an item is typed against (`?class1`) is itself an instance of a metaclass (`wd:Q19478619`), like *type of building*, then the item is an instance of a proper class-of-classes, not a punning entity, and should not be counted as such:

```
SELECT DISTINCT ?item WHERE {
  ?item wdt:P31 ?class1 .
  ?item wdt:P279 ?class2 .
  ?class1 wdt:P31/wdt:P279* wd:Q19478619 .
}
```

This removes **1k** classes (**4.7%**) and **2.4M** items (**56.5%**) from the census, leaving **22k** genuine punning classes and **1.8M** genuine punning items, which we use as the population for the rest of this analysis. Applying the same exclusion, we finally count how many of these genuine punning items have at least one instance:

```
SELECT DISTINCT ?item WHERE {
  ?item wdt:P31 ?class1 .
  ?item wdt:P279 ?class2 .
  ?instance wdt:P31 ?item .
}
```

Here, we get only **38k** punning items, meaning that around **98%** of punning items are in fact empty classes.

2.2. Patterns

The analysis of the **22k** distinct classes populated by punning items reveals three different patterns:

- **Pattern 1 (phantom classes).** The **98%** punning items that have no instance cover **16k** classes. Most come from bulk imports of external classification systems, where every specific term is modelled as both an instance and a subclass of its parent. Items related to biology or chemistry (e.g., *protein*, *gene*, *chemical element*) contribute to the vast majority of these items. An example is *hydroxyacylglutathione hydrolase*, that is P31 and P279 of *protein* but has no instance. Other typical examples are colors and food. For instance, *mauve* is P31 and P279 of *colour*, but no Wikidata item has ever been typed as an instance of *mauve*; the same holds for *tzatziki* under *food*. The dual typing P31 and P279 is questionable in these cases.
- **Pattern 2 (true punning items).** The second pattern concerns classes that, unlike phantom classes, actually have both instances and properties. One example is *Aircraft*, which has properties at the concept level (weight, width, etc.), and which has individual instances (“the Boeing airplane that crashed on that date”). Another is *Ocean liner*, characterised by properties such as tonnage and passenger capacity, while a specific ship such as the *RMS Titanic* is recorded as its own item, with its own history. This case covers around **6k** classes with **460k** items.
- **Pattern 3 (temporal/geographic series).** The last pattern correspond to temporal/geographic series of an event or an organization. It concerns around **1k** classes and **69k** items. Items corresponding to recurring events, such as *Nobel Prize in Physics* or *Prague Pride*, require dual typing for operational reasons: the series is an individual because it carries Class properties (e.g., founding year, awarding body) that do not propagate to editions; it is also a class because it contains the individual editions of the event. The same pattern concerns geographic and administrative hierarchies (e.g., electoral units, local administrations).

3. Use Case

If a statement has a class as object, it usually means that there exists an instance of that class that functions as the object of the statement. For example, in `:SteveJobs :speaks :English` (where `:English` is a class), it is sufficient to instantiate `:English` and to say that `:SteveJobs :speaks :SteveJobsEnglish`. This is in fact the mechanism that YAGO already uses when the object is a class [4]. The same cannot work if the subject is a class, as we will now show.

We design a prototypical use case around the example of the iPhone 12. This item can be seen both as a class and as an individual: **As a class**, it is the class of all iPhones 12. Its instances are individual iPhones, such as, e.g., John’s phone. **As an instance**, iPhone is an instance of the class of smart phone models. Beyond a mere philosophical distinction, different types of properties attach to the item, depending on which role it takes: **Class-properties** attach to the item itself, and are not propagated to the instances of the item. For example, the iPhone 12 has won the *iF Design Award 2021*, and this is a property of the brand, and not of every single iPhone 12. **Instance-properties**, in contrast, apply to every instance of the class. For example, the iPhone 12 has a weight of 164 grams, and this property is inherited to every instance of this phone. More formally, let us consider the following statements:

```
:iPhone12      rdfs:subClassOf   :iPhone
:iPhone12      :wonAward        :iFDesignAward2021 # class-level
:iPhone        :operatingSystem :iOS           # instance-level, all iPhones
:iPhone12      :weight          "164g"           # instance-level, all iPhones 12
:JohnsPhone    rdf:type         :iPhone12        # an instance
:expensiveBecause rdf:type      owl:ObjectProperty # Each phone is expensive due to
:expensiveBecause owl:propertyChainAxiom ( rdf:type :wonAward ) # the award of its model
:ringTone      rdfs:domain      :iPhone
:MarysPhone    :ringTone        :DingDing
```

We have the following desiderata:

Desideratum 1: Instance-properties propagate. The weight of John's iPhone is given at the level of the iPhone 12, i.e., the query `SELECT ?w WHERE :JohnsPhone :weight ?w` should return 164g.

Desideratum 1b: Instance-properties propagate across the taxonomy. The operating system of John's iPhone is given at the level of the iPhone, i.e., two levels up in the taxonomy. The query `SELECT ?w WHERE :JohnsPhone :operatingSystem ?w` should return :iOS.

Desideratum 2: Class-properties can be attached. By this, we mean not just that it is possible to say that `iPhone12 :wonAward :iFDesignAward2021`, but also to reason on a true punning item (Section 2.2) across both of its roles. In our example, we posit that each individual iPhone 12 (each instance) is expensive because of the award that the iPhone 12 (the class) won. That means: The query `SELECT ?w WHERE :JohnsPhone :expensiveBecause ?w` should return :iFDesignAward2021.

Desideratum 2b: Class-properties do not propagate. The :wonAward value of the iPhone 12 value must *not* propagate to :JohnsPhone. That is, the result of the query `SELECT ?a WHERE :JohnsPhone :wonAward ?a` shall be empty.

Desideratum 3: Domain and range constraints. The domain and range constraints of properties are respected. In our example, the query `SELECT ?w WHERE :MarysPhone rdf:type ?w` should return :iPhone. This answer is to be deduced from the fact that the domain of :ringTone are (in our somewhat contrived example) iPhones.

Desideratum 4: Transitivity of type and subclass relations. An instance of a class shall automatically be an instance of all of its superclasses. In other words, `SELECT ?w WHERE :JohnsPhone rdf:type ?w` shall return :iPhone12 and :iPhone.

Importantly, the queries shall return their answers without any need for mentioning intermediate classes in the query. The result shall just be deduced automatically through the semantics of the formalism. This means that any solution must be able to materialize the properties it infers as ordinary triples, so that they can be retrieved with standard SPARQL queries rather than requiring complex navigation.

4. Candidate Solutions

4.1. YAGO's current solution

YAGO separates all items into either individuals or classes. Every item that has a P279 property is considered a class, unless it has one of several manually defined properties. For example, any item with a manufacturer property becomes an instance. This has the effect of making commercial products instances. In our example, the iPhone becomes an individual.

Demoting concept-like entities to individuals and dropping their class role breaks nearly all desiderata: Instance properties cannot propagate, because there cannot be any instances (Desideratum 1 and 1b). For the same reason, domain and range constraints are not possible (Desideratum 3), and transitivity (Desideratum 4) cannot happen. Only the simple attachment of properties to classes works, but no reasoning is possible (Desideratum 2).

When an item becomes a class, no properties can be attached to it at all. Thus, no properties can exist (Desiderata 1, 1b, and 2), and none can propagate. Only transitivity and domain and range constraints work. This solution is thus far from perfect.

4.2. OWL Annotation Properties

As a minimal extension of YAGO's current practice, properties on classes could be declared as `owl:AnnotationProperty`. OWL 2 DL permits such properties on classes, precisely because reasoners ignore annotation properties entirely. Annotation properties participate in no entailment, so :operatingSystem and :weight are never inferred for :JohnsPhone (Desiderata 1 and 1b violated).

Class properties can be attached, but it is not possible to reason on them, i.e. Desideratum 2 is not fulfilled. Desideratum 2b is trivially fulfilled, because no reasoning happens on class properties. Domain and range constraints work as intended (Desideratum 3), albeit only for the domain. We cannot restrict the range of `:wonAward`, because no reasoning can happen on annotation properties. The transitivity of type and subclass relations is preserved (Desideratum 4), because `rdf:type/rdfs:subClassOf` entailment is untouched by which properties are marked as annotations. This is the mechanism YAGO already uses for the handful of properties currently permitted on class nodes (`rdfs:label`, `schema:sameAs`); extending the list adds metadata storage, not reasoning. Thus, this solution does not tick our desiderata.

4.3. OWL Punning

OWL 2 actually offers a native method for punning: The same IRI may be declared both a class and an individual, exactly as `:iPhone12` would need to be. But this is permitted only under a strict semantic separation: the class reading and the individual reading of the same IRI are treated as two unrelated elements of the model, so no fact can flow between the two roles [6, 7].

```
:iPhone12 rdf:type owl:Class .      # class reading
:iPhone12 rdf:type owl:NamedIndividual . # individual reading, unrelated
:iPhone12 :weight "164g" .           # asserted on the individual reading only
:JohnsPhone rdf:type :iPhone12 .     # typed against the class reading
```

Since the two readings share nothing but the IRI, no propagation (Desideratum 1 and 1b) can be derived: `:weight` is asserted on the individual reading of `:iPhone12`, but `:JohnsPhone` is typed against its class reading, and OWL 2 semantics guarantees no entailment crosses that boundary. Desideratum 2 fails for the same reason: no deduction can involve an item in both readings. The non-propagation deduction (Desideratum 2b) holds, but just because nothing propagates, so `:wonAward` cannot leak either.

Domain and range constraints (Desideratum 3) and transitivity of type and subclass relations (Desideratum 4) are preserved for the class reading in isolation, but this is of little help, since the class reading cannot carry the Class facts (`:wonAward`) that motivated punning in the first place — those must live on the individual reading, which participates in no subsumption hierarchy at all. This avoids rather than resolves the underlying modelling conflict: many properties are used identically regardless of role (an `rdfs:label` or properties such as `schema:foundationYear` of an `Event` apply equally to the class and to the individual), so treating the two readings as entirely unrelated departs from how such entities are actually described in practice, not merely as a technical inconvenience. In summary, OWL 2 punning is a syntactic permission, not an inference mechanism.

4.4. RDFS (Subproperty with Domain/Range)

In RDFS, properties are attached directly to the class node, and punning is unrestricted:

```
:iPhone :operatingSystem :iOS .
:iPhone12 rdfs:subClassOf :iPhone .
:iPhone12 :weight "164g" .
```

RDFS entailment rule *rdfs9* propagates `rdf:type` through the subclass chain (so `:JohnsPhone` is inferred to be of type `:iPhone12` and `:iPhone`), but no RDFS rule propagates property *values*: both `SELECT ?os WHERE { :JohnsPhone :operatingSystem ?os }` and the equivalent query for `:weight` return empty, even though the class node two hops away carries the value. Both propagation deductions (Desideratum 1 and 1b) are recoverable only by navigating `rdfs:subClassOf` manually with a property-path query, which fails the materialisation requirement. Reasoning is not possible at all, so Desideratum 2 fails. The non-propagation deduction (Desideratum 2b) holds, but trivially: `:wonAward` is never inherited because *nothing* is, not because RDFS distinguishes Instance from Class properties. Transitivity of type and subclass relations (Desideratum 4) is preserved natively (that is exactly what *rdfs9* provides). Domain and range constraints (Desideratum 3) are also supported, albeit in a deductive way rather than as a constraint: RDFS domain/range declarations infer new type memberships but cannot detect an inconsistent assertion

the way OWL DL restrictions can. The central problem (propagating `:operatingSystem` and `:weight`) remains unsolved, making this solution not ideal.

4.5. SHACL (NodeShape with `sh:rule`)

SHACL derives both propagation deductions (Desideratum 1 and 1b) procedurally: a forward-chaining rule fires for every node of a target class and materialises a named-IRI triple.

```
:OSRule a sh:NodeShape ;
  sh:targetClass :iPhone ;
  sh:rule [ a sh:SPARQLRule ;
    sh:construct
      "CONSTRUCT { $this :operatingSystem ?o }
        WHERE { :iPhone :operatingSystem ?o }" ] .
```

Tested with `pyshacl 0.31.0`, it correctly derives `:JohnsPhone :operatingSystem :iOS` and `:JohnsPhone :weight "164g"`, materialised exactly as desired. A similar rule could achieve the deduction needed by Desideratum 2. The non-propagation deduction (Desideratum 2b) is achieved simply by not writing a rule for `:wonAward`: since no rule ever targets that property, it can never leak to an instance. The transitivity of type and subclass relations (Desideratum 4), as well as domain and range constraints (Desideratum 3) are both preserved, because SHACL rules run alongside, not instead of, the base RDFS/OWL entailment: a separate NodeShape can still validate domain and range, and `rdfs:subClassOf/rdf:type` entailment is untouched. The mechanism is procedural, not ontological, though: a SHACL rule is a forward-chaining production rule executed by a separate rule engine, not a Description Logic axiom with model-theoretic semantics, so its consequences depend on the engine that runs it rather than on standard OWL entailment. The solution thus ticks all desiderata, albeit in a rather expensive way.

4.6. SKOS

SKOS abandons the class/individual distinction entirely: Every punning entity becomes a `skos:Concept`, and the taxonomy uses `skos:broader/narrower` instead of `rdfs:subClassOf/rdf:type`:

```
:iPhone12 a skos:Concept ;
  skos:broader :iPhone ;
  :weight "164g" .
```

Technically, all items thus become individuals. This dissolves the class/individual distinction by construction, but at a steep price: No native RDFS/OWL deductions are possible any more. Hence, all deductions would have to be implemented outside SKOS, for example by help of OWL property chain axioms (`owl:propertyChainAxiom`). The solution is thus possible, but at the expense of giving up upon, and reconstructing manually, the entire RDFS reasoning mechanism. The domain and range constraints, in particular, would need property restriction classes.

4.7. Ontology Design Patterns

A different family of approaches addresses the dual role of an entity and the two kinds of properties on it as modelling conventions rather than through an inference mechanism. The W3C's own catalogue of patterns for representing classes as property values [8] documents five such conventions — including a parallel instance hierarchy, essentially our kind layer below — but explicitly stops short of a class that must simultaneously carry descriptive properties and participate in a type hierarchy, which is exactly our problem. The powertype pattern [9] comes closer: a class *P* is a powertype of *C* if every instance of *P* is itself a subclass of *C* (e.g. a designated class *PhoneModel* whose instances, such as `:iPhone12`, are subclasses of *Phone*). This gives a principled place to declare dual-role entities, but a powertype declaration is silent on inference: it does not say which properties of a powertype instance should

propagate to its own instances, so neither propagation deduction (Desideratum 1 and 1b) is derived. The non-propagation deduction (Desideratum 2) for `:wonAward` holds only in the same trivial sense as for RDFS and SKOS above: nothing propagates, so nothing is ever asserted on `:JohnsPhone` by mistake. Since the design patterns piggybacks on OWL inference, deductions across the class-instance separation (as requested by Desideratum 2) are not possible. Domain and range constraints (Desideratum 3) and transitivity of type and subclass relations (Desideratum 4) are both preserved, since a *powertype* declaration does not touch the base RDFS/OWL entailment, it only adds a modelling convention on top of it. Ontology design patterns therefore organise dual-role entities structurally, but do not by themselves derive any of the three deductions.

4.8. Kind Layer

Another approach is a *kind-layer model* that removes the class/individual distinction from the lower taxonomy altogether, replacing `rdf:type/rdfs:subClassOf` with two navigational predicates: `:hasCategory` (individual $\rightarrow$ kind) and `:subCategoryOf` (kind $\rightarrow$ parent kind). `rdf:type` is kept only for coarse top-level schema typing needed for domain checks. This solution is inspired by the way Wikidata deals with taxons (animals): One taxon can be declared a *parent taxon* (P171) of another one. In this way, one can simulate a class hierarchy among individuals. In our use case, this solution yields:

```
:JohnsPhone  rdf:type      schema:Product .
:JohnsPhone  :hasCategory  :iPhone12 .
:iPhone12    :subCategoryOf :iPhone .
:iPhone12    :weight       "164g" .
:iPhone12    :wonAward     :iFDesignAward2021 .
```

Since no triple is ever materialised on the instance node, `SELECT ?w WHERE { :JohnsPhone :weight ?w }` returns empty: both propagation deductions (Desideratum 1 and 1b) require an explicit multi-hop navigation. The same goes for Desideratum 2: One can find that John's iPhone model has won an award, but only by chaining joins. The non-propagation deduction (Desideratum 2b) is worse than merely unsatisfied: it is actively at risk, because nothing in the model distinguishes a property that should propagate from one that should not. The same navigation query written for `:weight` (which is: follow `:hasCategory` then `:subCategoryOf` zero or more times and read off the property) would, if applied to `:wonAward`, incorrectly return `:iFDesignAward2021` for `:JohnsPhone`. Avoiding this requires the query author to already know, from outside the model, which properties are individuals and which are classes. Domain and range constraints (Desideratum 3) are preserved only at the coarse, top-level `rdf:type` (e.g. `schema:Product`), not at the fine-grained level of `:iPhone12` or `:iPhone`, and transitivity of type and subclass relations (Desideratum 4) is not real `rdfs:subClassOf` entailment either, only a `:subCategoryOf*` property-path query. To address these weaknesses, the approach could be combined with OWL property chain axioms to make the deductions of Desideratum 1, 1b, 2, and 4, and with property restriction classes to model domain and range constraints (Desideratum 3). The axioms could also distinguish between instance- and class-properties (Desideratum 2b). However, this would mean an ad-hoc combination of OWL and SKOS, where propagation happens across different mechanisms, depending on the depth of the taxonomy.

4.9. OWL General Concept Inclusions

A *General Concept Inclusion (GCI)* is a Description Logic axiom $C \sqsubseteq D$, where D may be a property restriction. For example, `iPhone  $\sqsubseteq$   $\exists$  operatingSystem.{iOS}` states that every instance of `iPhone` has that operating system. The axiom lives in the TBox and fires automatically for every ABox individual of the class; in RDF/OWL it is serialised as `rdfs:subClassOf` with a blank-node `owl:Restriction`, and an OWL RL reasoner materialises the derived triples:

```
:iPhone rdfs:subClassOf [
  a owl:Restriction ;
```

```

owl:onProperty :operatingSystem ;
owl:hasValue  :iOS ] .

:iPhone12 rdfs:subClassOf [
  a owl:Restriction ;
  owl:onProperty :weight ;
  owl:hasValue  "164g" ] .
:iPhone12 :wonAward :iFDesignAward2021 . # plain triple

```

We tested this encoding with the owlrl 0.6 reasoner. It derives `:JohnsPhone :operatingSystem :iOS` (two hops, via `:iPhone12  $\sqsubseteq$  :iPhone`) and `:JohnsPhone :weight "164g"` (one hop) as named-IRI triples reachable by a single triple-pattern query (Desideratum 1 and 1b), while `:wonAward` remains a plain triple on `:iPhone12` and is never propagated (Desideratum 2b): the choice between a GCI and a plain triple is exactly the decision the modeller makes per property, propagating the first two and withholding the third. GCIs propagate through arbitrarily long `rdfs:subClassOf` chains, so both propagation deductions hold at any hierarchy depth, and OWL 2 punning allows the same IRI to be both a class and an individual. Domain and range constraints (Desideratum 3) and transitivity of type and subclass relations (Desideratum 4) are both preserved, since GCIs are additional axioms rather than a change to the underlying data model. This solution thus ticks all desiderata, except for Desideratum 2: OWL GCIs cannot reason on items that are classes and instances at the same time.

4.10. Comparison

Our analysis is summarized in Table 1.

	Punning	GCI	RDFS	SHACL	SKOS	Annot.	ODP	Kind
Instance properties propagate (1, 1b)	×	✓	×	(✓)	(✓)	×	×	(✓)
Class property allows reasoning (2)	×	×	×	(✓)	(✓)	×	×	(✓)
Class property does not propagate (2b)	✓	✓	✓	✓	(✓)	✓	✓	(✓)
Domain and range constraints preserved (3)	✓	✓	✓	✓	(✓)	(✓)	✓	(✓)
Transitivity of type/subclass preserved (4)	✓	✓	✓	✓	(✓)	✓	✓	(✓)

Table 1

Evaluation of the solutions on the use case. ✓ = satisfied; × = not satisfied; (✓) = satisfied with additional machinery.

We can conclude that no single solution satisfies all desiderata out of the box. SHACL can do basically everything, but needs to be equipped with SPARQL queries, which means expensive reasoning outside the reasoner. OWL GCIs are powerful, but fail on the central criterion of reasoning on true punning items. SKOS and the Kind layer are ad hoc solutions, which can express everything, but destroy the RDFS semantics. A combination with OWL axioms could recover these semantics, but at the price of bypassing the classical mechanism entirely, or running in parallel to it.

5. Conclusion

We showed empirically that punning affects a large number of classes in Wikidata, and that it decomposes into three structurally distinct patterns: phantom classes that need no dual role at all, latent classes whose class role is exercised only when individual specimens are represented, and temporal/geographic series that need both roles simultaneously. In our work, we do not resolve the ambiguity of these cases, i.e., we do not decide or suggest whether an entity should be an individual or a class. Rather, we study how its class-like characteristics and its individual-like characteristics can both be represented.

We find that YAGO’s current binary class/individual assignment cannot represent both characteristics, and we argue that this is not a YAGO-specific curiosity: Any tool built on Wikidata that needs to tell a genuine class from a data-entry artefact faces the same problem. Against the iPhone 12 use case and

the domain/range and transitivity requirements derived from it, we found that lightweight, already-available mechanisms (RDFS, OWL annotation properties) are cheap but cannot propagate Instance properties at all; that ontology design patterns give dual-role entities a principled place in a schema but do not by themselves derive any propagation behavior; that SHACL matches OWL’s inferential power procedurally, at the cost of hand-maintained rules; that SKOS removes the class/individual split altogether, and would need a recreation of the RDFS semantics through additional axioms; that a kind-layer model sits uneasily between RDFS and SKOS; and that OWL General Concept Inclusions tick nearly all desiderata properly, but cannot reason on true punning items.

This leaves the question of reasoning on punning as an interesting research avenue with potentially large impact for both YAGO and Wikidata. For YAGO, a solution would mean a huge gain in facts that currently cannot be imported from Wikipedia because they concern punning items. For Wikidata, any progress in formalizing the distinction between classes and instances (e.g., powertype-like patterns to organise dual-role items, and GCI-style axioms to make the right facts propagate) would let downstream tools stop guessing which P279 edges are load-bearing. More fundamentally, whether a property applies to every instance of a class or merely can apply to some of its instances could itself be treated as a property of that property, rather than decided ad hoc per use case.

One possible avenue for future work is to leave the strict distinction between classes and instances behind, and to implement all the desired functionality as rules. Indeed, all required patterns of our use case in Section 3 take the form of Horn rules. Such rules could then be materialized at query-time by a rule-based reasoning engine such as Nemo [10], RDFox [11], or Bash Datalog [12].

We hope that these insights can spur the further development of both YAGO and Wikidata.

Acknowledgments

This work was carried out as part of a research internship at Institut Polytechnique de Paris. We sincerely thank Markus Krötzsch for his valuable feedback on Sections 3, 4.3, and 4.9.

Declaration on Generative AI

Generative AI tools were used to polish the language and refine the phrasing of this manuscript.

References

- [1] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider (Eds.), *The Description Logic Handbook: Theory, Implementation and Applications*, Cambridge University Press, 2003.
- [2] I. Horrocks, U. Sattler, S. Tobies, Reasoning with individuals for the description logic SHIQ, in: *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, volume 1831 of *LNAI*, Springer, 2000, pp. 482–496.
- [3] D. Vrandečić, M. Krötzsch, Wikidata: A free collaborative knowledgebase, *Communications of the ACM* 57 (2014) 78–85.
- [4] F. M. Suchanek, M. Alam, T. Bonald, L. Chen, P.-H. Paris, J. Soria, YAGO 4.5: A large and clean knowledge base with a rich taxonomy, in: *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR ’24*, ACM, New York, NY, USA, 2024. doi:10.1145/3626772.3657876.
- [5] T. P. Tanon, G. Weikum, F. M. Suchanek, YAGO 4: A reasonable knowledge base, in: *Proceedings of the 17th Extended Semantic Web Conference (ESWC)*, Springer, 2020, pp. 583–596.
- [6] B. Motik, P. F. Patel-Schneider, B. Cuenca Grau, *OWL 2 Web Ontology Language Direct Semantics (Second Edition)*, W3C Recommendation, W3C, 2012.
- [7] W3C OWL Working Group, Punning, W3C OWL Wiki, 2007. <https://www.w3.org/2007/OWL/wiki/Punning>.

- [8] N. F. Noy, A. Rector, Representing Classes As Property Values on the Semantic Web, W3C Working Group Note, W3C, 2005. <https://www.w3.org/TR/swbp-classes-as-values/>.
- [9] C. Gonzalez-Perez, B. Henderson-Sellers, A powertype-based metamodeling framework, *Software and Systems Modeling* 5 (2006) 72–90.
- [10] A. Ivliev, L. Gerlach, S. Meusel, J. Steinberg, M. Krötzsch, Nemo: Your friendly and versatile rule reasoning toolkit, in: *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2024, pp. 743–754.
- [11] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, J. Banerjee, Rdfox: A highly-scalable RDF store, in: *ISWC*, 2015.
- [12] T. Rebele, T. P. Tanon, F. M. Suchanek, Bash Datalog: Answering Datalog Queries with Unix Shell Commands, in: *ISWC*, 2018.