Datatype Evolution on RDF Ontologies

Anastasia Analyti[1] and Ioannis Pachoulakis[2]

[1] Institute of Computer Science, FORTH-ICS, Crete, Greece
[2] Dept. of Informatics Engineering, TEI Crete, Crete, Greece
analyti@ics.forth.gr, ip@ie.teicrete.gr

Abstract. Semantic Web ontologies are not static but evolve as the understanding of the domain (or the domain itself) grows or evolves. This evolution is usually independent of the ontological instance descriptions which are stored in Knowledge Bases (KBs). Therefore, the effects of an ontology update on instance descriptions which are migrated to a new ontology version should be studied. In this paper, we are interested in the effects of the migration of literal-valued instance triples of an RDF KB to a new ontology version. Note that the datatype of the property of such an instance description may have evolved in the new ontology version, e.g. from integer to decimal or from float to double. We manage the quality of these migrated instance triples and design flexible interactive methods through which the curator of the KB can intervene to correct the "inaccuracies" of the literal values that are incurred by the migration process. In addition, specialized algorithms are presented, which are highly efficient, since their time and space complexity is linear w.r.t. the size of the literal-valued instance triples of the KB. Our work is particularly important to e-Science applications that are commonly curated and have many datatype properties.
Keywords: Datatype evolution, literal-valued instance triples, quality management, Knowledge Base curation.

1 Introduction

XML is the dominant language for information exchange. The XML Schema Definition Language (XSD) provides an extended set of datatypes [23] which is particularly useful to e-Science applications that have many datatype properties. However, new requirements may arise in an application that lead to ontology evolution[3], including updates on the ranges of several datatype properties. Moreover, the integration of heterogeneous sources may also require such updates. However, ontology evolution affects the "accuracy" of the literal-valued instance triples of the application. For example, consider the literal-valued instance triple (*John hasWeight* "72"^^*xsd*:*integer*) which was stated according to the original ontology, where *hasWeight* is a property with datatype *xsd*:*integer*. If the datatype of the property *hasWeight* in the new ontology version is *xsd*:*decimal* then 72 may not be the most accurate weight value for *John*, and the most accurate one may be in the range $(71, 73)$[4].

In this paper, we deal with the case of *XSD datatype evolution*, in the general context of ontology evolution. XSD datatypes are divided into primitive XSD

[3] Here, the term *ontology* refers to schema information.
[4] In this paper, the interval (a, b) denotes the set of decimal values that are greater than a and less than b. Similar is the case for the open-closed, closed-open, and closed-closed intervals $(a, b]$, $[a, b)$, and $[a, b]$, respectively.

datatypes and derived XSD datatypes, the latter being derived from the primitive XSD datatypes or other derived XSD datatypes through facet restriction [23]. We shall denote the *direct evolution* from a primitive datatype d to a primitive datatype d' by $d \rightarrow d'$. We support the following datatype direct evolutions: *float* $\rightarrow$ *double*, *gYear* $\rightarrow$ *gYearMonth*, *gYearMonth* $\rightarrow$ *date*, *date* $\rightarrow$ *dateTime*, *gMonth* $\rightarrow$ *gMonthDay*, *hexBinary* $\rightarrow$ *base64Binary*, and *base64Binary* $\rightarrow$ *hexBinary*. In addition, an XSD datatype d_1 can *evolve* to another XSD datatype d_2, denoted by $d_1 \Rightarrow d_2$, if (i) each value of d_1 can be expressed to a value of d_2 and (ii) if d_1 is recursively derived from a primitive datatype d'_1 and d_2 is recursively derived from a primitive datatype d'_2, it holds that either $d'_1 = d'_2$ or $d'_1 \rightarrow^* d'_2$ [5].

Some examples of datatype evolution appear in Table 1[6]:

Case	Datatype evolution	Notes	Property example
Linearly ordered infinite sets	integer ⇒ decimal	transformation to a more precise datatype	hasWeight
Intervals over linearly ordered finite sets	$\{0, 1, 2, ..., 100\} \Rightarrow \{0, 1, 2, ..., 200\}$	expansion of the datatype range	hasAge
Linearly ordered finite sets to linearly ordered infinite sets	{1,2,3,...,200} ⇒ [0,300]	expansion of the datatype range	hasWeight
Simple datatype to complex datatype	gYear ⇒ date	transformation to a more informative datatype	hasBirthday
Unordered enumerated datatypes	{white, black} ⇒ {white, black, grey}	expansion of unordered enumerated values	hasColour
Ordered enumerated datatypes	{0,1,2,3} ⇒ {0, 0.5, 1, 1.5, 2, 2.5, 3}	expansion of ordered enumerated values	hasGrade
Unordered datatypes	string with maxLength=20 ⇒ string with maxLength=30	expansion of datatype range	hasName
Linearly ordered infinite sets	decimal with fractionDigits=1 ⇒ decimal with fractionDigits=2	transformation to a more precise datatype	hasWeight

Table 1. Examples of datatype evolution

To express and manage the uncertainty incurred by literal-valued instance triple migrations, we present a general model which formalizes the problem and details the related inferences. The inferred information is represented in the form of *compact possible datatype tuples* (representing a possibly infinite set of possibilities), one for

[5] By $\rightarrow^*$, we denote the transitive closure of the relation $\rightarrow$ between primitive XSD datatypes.

[6] A decimal with fractionDigits=x is a decimal with maximum number of fraction digits equal to x [23].

each migrated literal-valued instance triple. The idea is further elaborated in Table 2[7].

Datatype evolution	Derived literal value possibilities	Property example
integer ⇒ decimal	72 → (71,73)	hasWeight
{0,1,2,...,100} ⇒ {0,1,2,...,200}	0 → 0, 1 → 1, ... 99 → 99, 100 → {100,101,102,...,200}	hasAge
{1,2,3,...,200} ⇒ [0,300]	1 → (0,2), 2 → (1,3), 200 → (199,300]	hasWeight
gYear ⇒ date	2011 → [2011-01-01, 2011-12-31]	hasBirthday
{white, black} ⇒ {white, black, grey}	white → {white,grey} black → {black,grey}	hasColour
{0,1,2,3} ⇒ {0, 0.5, 1, 1.5, 2, 2.5, 3}	$0 \rightarrow \{0, 0.5\},\ 1 \rightarrow \{0.5, 1, 1.5\},$ $2 \rightarrow \{1.5, 2, 2.5\},\ 3 \rightarrow \{2.5, 3\}$	hasGrade
string with maxLength=20 ⇒ string with maxLength=30	"Dom. Theotocopoulos" → any string of maxLength=30 and minLength=21 except "Dom. Theotocopoulos", e.g. "Domenico Theotocopoulos"	hasName
decimal with fractionDigits=1 ⇒ decimal with fractionDigits=2	23.2 → any decimal in (23.1,23.3) with fractionDigits=2	hasWeight

Table 2. Examples of derived literal value possibilities for the datatype evolution cases of Table 1

The ability for this kind of inference and the management of the resulting inferred information can be exploited in a variety of contexts, such as:

- Ontology evolution, where the XSD datatype of a property evolves to a new XSD datatype, as described earlier.
- Centralized repositories that integrate scientific data from various heterogeneous sources (these can store the possibilities resulting from the integration of the different XSD datatypes of the same property and exploit this information in their query services).
- Exchange formats which prescribe the precision of the contained values.

After computing the set of compact possible datatype tuples, we focus on the exploitation of such tuples aiming to increase the accuracy of the literal-valued instance triples. In particular, we propose a curation process through which the curator of the RDF Knowledge Base (KB) can view the computed possibilities and select the most accurate literal values for the literal-valued instance triples whose accuracy was affected by the migration. It should be made clear that the curation process is optional and the curator can improve on the accuracy of any subset of affected literal-valued instance triples. As ontology evolution continues, the computed possibilities are updated and the curation process can start at any stage of the ontology evolution process.

[7] The arrow → in the table is read as *has possibilities* (after the migration). To understand the first line of the table, consider our introductory example. Additionally, with the interval [2011-01-01, 2011-12-31], we represent all dates from 2011-01-01 to 2011-12-31, where dates are written in the form Year-Month-Day.

To achieve our goals, several algorithms are presented, which are highly efficient, since their time and space complexity is linear w.r.t the size of the literal-valued instance triples of the KB. In particular, we present algorithms that initialize the set of compact possible datatype tuples in the original ontology and derive the new set of compact possible datatype tuples, appropriate to the evolved ontology.

Note that the current trend in e-Science is to represent scientific data in Semantic Web languages using domain specific schemas (ontologies expressed in RDF) and publish them as Linked Open Data [12]. Since RDF relies on XSD datatypes, our framework captures the requirements of both RDF and Linked Open Data. To the best of our knowledge, this is the first work that reasons on the accuracy of the XSD-typed literal values of property instance triples that is affected by datatype evolution.

The remaining of this paper is organized as follows: Section 2 contains a motivating example of the curation process. Section 3 presents background information for the XSD datatypes. Section 4 defines the valid RDF KBs and datatype evolution. Section 5 defines datatype backwards compatible KB evolution and the set of possible and false datatype instance tuples. Section 6 presents specialized algorithms to derive the original set of compact possible datatype tuples and also the new set, which is derived based on the old one during ontology evolution. Section 7 presents a life cycle management process through which KB curators can improve on the accuracy of the XSD-typed literal values of property instance triples that are affected by ontology evolution. Section 9 presents related work and, finally, Section 10 presents our conclusions and possible directions for further research. Due to paper size limitations an electronic Appendix is provided at `http://www.ics.forth.gr/~analyti/Papers_2/Datatype_Evolution_Appendix.pdf`. Appendix A presents the supported primitive XSD datatypes and their facets. Appendix B presents a number of constraints that XSD datatype facet values should satisfy. Appendix C presents the derived built-in XSD datatypes. Appendix D presents several algorithms that for readability reasons are not included in the main paper. Appendix E presents the proofs of all Lemmas and Propositions that appear in the main paper. Finally, Appendix F contains additional examples.

2 Motivating Example of the Curation Process

To understand the curation process and the value of the computed possibilities, assume that the original KB contains the literal-valued instance triples (*John hasWeight* "72"^^*xsd*:*integer*) and (*Mary hasWeight* "23"^^*xsd*:*integer*). Assume that in the next ontology version the range of the property *hasWeight* changes from *xsd*:*integer* to *ex*:*decFracDigits1* (meaning the set of decimals with fractionDigits=1). Then, the two instance triples are automatically transformed to (*John hasWeight* "72"^^*ex*:*decFracDigits1*) and (*Mary hasWeight* "23"^^*ex*:*decFracDigits1*). Now, the computed possibilities for (*John hasWeight* "72"^^*ex*:*decimalFracDigits1*) are all decimals in $(71, 73)$ with fractionDigits=1 and the computed possibilities for (*Mary hasWeight* "23"^^*ex*:*decFracDigits1*) are all decimals in $(22, 24)$ with fractionDigits=1. Then, the curator inspecting the two instance triples and their computed possibilities may decide[8] to replace (*Mary hasWeight* "23"^^*ex*:*decFracDigits1*) by the more precise instance triple (*Mary hasWeight* "23.2"^^*ex*:*decFracDigits1*). Now the computed possi-

[8] He possibly has no more specific information about the weight of John.

bilities for (*Mary hasWeight* "23.2"^^*ex*:*decFracDigits1*) are updated to {23.2}. That is, 23.2 is the most precise decimal value with one fraction digit for the weight of Mary. Assume that in the next ontology version the range of the property *hasWeight* changes from *ex*:*decFracDigits1* to *ex*:*decFracDigits2* (meaning the set of decimals with fractionDigits=2). Now, the computed possibilities for (*John hasWeight* "72"^^ *ex*:*decFracDigits2*) are all decimals in $(71, 73)$ with fractionDigits=2, while the computed possibilities for (*Mary hasWeight* "23.2"^^*ex*:*decimalFracDigits2*) are all decimals in $(23.1, 23.3)$ with fractionDigits=2. The curation process may now start again. Obviously, even if the curator may not have more precise information about the weight of John and Mary, the computed possibilities provide useful information to the user.

Though the curation process is a triple-by-triple fixing activity, this does not mean that the curator has to do it by hand. There may exist an RDF file containing literal-valued instance triples, which is automatically parsed (one triple at the time), and if it contains more precise information than that already existing in the KB it replaces it and the possibilities of the new tuple are computed. Checking if a literal-valued instance triple contained in the file is more precise than one (say t) contained in the KB requires a check over the computed possibilities of t.

3 Background

A (Web) *vocabulary* V is a set of URI references and/or literals (plain or typed). A plain literal is a string "s", where s is a sequence of Unicode characters, or a pair of a string "s" and a language tag t, denoted by "s"@t. A typed literal is a pair of a string "s" and a datatype URI reference u, denoted by "s"^^u. For example, "27"^^*xsd*:*integer* is a typed literal.

We denote the set of all URI references by *URI*, the set of all plain literals by $\mathcal{PL}$, the set of all typed literals by $\mathcal{TL}$, and the set of all literals by $\mathcal{LIT}$. We consider a set $\mathcal{BL}$ of blank nodes, such that the sets $\mathcal{BL}$, *URI*, $\mathcal{LIT}$ are pairwise disjoint.

Let V be a vocabulary. An *RDF triple* over V is a triple $(s\ p\ o)$, where $p \in V \cap URI$ is called *property*, $s \in (V \cap URI) \cup \mathcal{BL}$ is called *subject*, and $o \in V \cup \mathcal{BL}$ is called *object*. An *RDF KB* K (for short, *KB*) is a set of RDF triples over some vocabulary V. For more details, see [19, 11].

In the RDF semantics specification [11], the *RDFS entailment* regime is defined. However, RDFS entailment does not support many useful datatypes. In fact, `rdf:XMLLiteral` is the only datatype that is supported. The RDF semantics specification also defines D *entailment* (datatype entailment), which extends RDFS entailment with the support of datatypes. Ter Horst [27] defines the notion of D^* *entailment*, which is also an extension of RDFS entailment, but semantically weaker that D entailment. In [5], a set of sound and complete rules is presented for D^* entailment and a set of sound but not complete rules for D entailment.

The definition of D entailment and D^* entailment is based on the notion of *datatype map*. Datatypes define sets of concrete data values (e.g., strings and integers). A *datatype* is a tuple $d = \langle L^d, V^d, V^d(.)\rangle$, consisting of:

- a *lexical space* L^d, which is a set of character strings (e.g., "0", "1", "01",..., in the case of an integer datatype), called the *lexical values* of d,
- a *value space* V^d, which is a set of values (e.g., the numbers 0,1,2,..., in the case of an integer datatype), and

– a *lexical-to-value mapping* $V^d(.)$, which is a surjective mapping from the lexical space to the value space (e.g., {"0" → 0, "1" → 1, "01" → 1, ...}, for an integer datatype).

A *datatype map* D is a partial mapping from URI references to datatypes, for which it holds that D(`rdf:XMLLiteral`) is the built-in XML literal datatype, as defined in the RDF specification [19, 11]. With $dom(D)$ and $ran(D)$, we denote the domain and range of D, respectively. Given a datatype map D, we say that a typed literal s^^u is *well-typed* if $u \in dom(D)$ and $s \in L^{D(u)}$. Additionally, we say that a typed literal s^^u is *ill-typed* if $u \in dom(D)$ and $s \notin L^{D(u)}$.

In this work, we consider a datatype map D, called *extended XSD* datatype map, that supports (i) most of the primitive XSD datatypes [23] (if fact these considered in [11]), and (ii) the datatypes derived by *facet-restriction* from the supported primitive XSD datatypes, i.e. by using facets on an existing datatype, so as to limit the number of possible values of the derived datatype (as considered in [2]). Thus, the supported XSD datatypes are divided into a set of *primitive datatypes*, denoted by XSD^{prim}, and a set of *derived datatypes*, denoted by XSD^{der}. We define $XSD = XSD^{prim} \cup XSD^{der}$. The supported primitive XSD datatypes are *decimal, float, double, dateTime, gYear, gYearMonth, date, gMonth, gMonthDay, gDay, time, boolean, hexBinary, base64Binary, string*, and *anyURI*. These datatypes are reviewed in Appendix A (see also [23]).

Each primitive XSD datatype d has an one-to-one mapping, called *canonical mapping*, $C^d : V^d \rightarrow L^d$ s.t. if $s \in L^d$ then it holds that $V^d(C^d(V^d(s)) = V^d(s)$, that is the value of the canonical representation of a literal coincides with the value of the literal. If $v \in V^d$ then $C^d(v)$ is called the *canonical value* of v according to datatype d.

Example 1. Consider the datatype *decimal*, though $V^{decimal}$("1.0") = $V^{decimal}$("+1") = $V^{decimal}$("1") = 1, it holds that the canonical value for the decimal 1 is "1" and not "1.0" or "+1", i.e. $C^{decimal}(1)$ = "1". Similarly, though V^{float}("1.00000001") = V^{float}("+1") = V^{float}("1") = V^{float}("1.0E1") = 1, the canonical value of float 1 is "1.0E1", i.e. $C^{float}(1)$ = "1.0E1". □

We define the set of *canonical values* of a datatype d, as follows: $C^d = C^d(V^d)$.[9]

Let $s \in L^d$, where $d \in \{string, anyURI, gYear, gYearMonth, date, gMonth, gMonthDay, gDay\}$. It holds that $C^d(V^d(s)) = s$.

We also support datatypes derived (based on facet-restriction) from the primitive XSD datatypes. *Constraining facets* are schema components whose values may be set or changed during derivation to control various aspects of the derived datatype. These facets are given a value as part of the derivation. Thus, we may have a chain of datatypes derived from a primitive XSD datatype. When a constraining facet F is associated with a primitive XSD datatype d [10] then F is also associated with all datatypes derived from d. If a datatype d' is directly derived from a datatype d through the specification of values for a set of facets FS then we say that each facet $F \in FS$ is a *declared facet* of d' and it holds $d'.F = f$, where f is the value of F.

If a datatype d is directly derived from another datatype d', we write $baseType(d) = d'$. We also say that d' is the *base type* of d. If $baseType(d) = d'$ then $L^d \subseteq L^{d'}$,

[9] Note that the notation C^d declares both a function and a set.

[10] This means that a datatype d' can be derived from d by setting a value f on the constraining facet F, i.e. $d'.F = f$.

$V^d \subseteq V^{d'}$, and $V^d(.)$ is the restriction of $V^{d'}(.)$ to L^d. Thus, if $s \in L^d$, $V^d(s) = V^{d'}(s)$. Let d be a datatype, we define $derFrom(d)$ to be the set $\{d\}$ union the set of datatypes that are directly or indirectly derived from d. Let d, d' be datatypes. If $d \in derFrom(d')$ and $d \neq d'$ then we also say that d' is an *ancestor type* of d. If $d \in derFrom(d')$ and $d' \in XSD^{prim}$ then we write $primAncestor(d) = d'$.

The canonical mapping of a derived datatype d, $C^d : V^d \rightarrow L^d$, is the restriction of the canonical mapping of $primAncestor(d)$ to V^d. Thus, if $x \in V^d$ then it holds that $C^d(x) = C^{primAncestor(d)}(x)$. Similarly to primitive datatypes, we define $C^d = C^d(V^d)$.

Let $d \in XSD$ and $s \in L^d$. We define the *canonical representation* of s according to datatype d, as follows: $L2C^d(s) = C^d(V^d(s))$. Note that if $primAncestor(d) = d'$ then $L2C^{d'}(s) = C^{d'}(V^{d'}(s)) = C^d(V^d(s)) = L2C^d(s)$. For example, $L2C^{float}(\text{“1.00000001”}) = \text{“1.0}E1\text{”}$ and $L2C^{integer}(\text{“}+1\text{”}) = L2C^{decimal}(\text{“}+1\text{”}) = \text{“1”}$.

Note that if $s \in L^d$, where $d \in \{string, anyURI, gYear, gYearMonth, date, gMonth, gMonthDay, gDay\}$, then $L2C^d(s) = s$.

Let d be a datatype. If the value space of $primAncestor(d)$ is ordered then the value space of d has the same ordering. If the value space of a datatype d is ordered then we say that $ordered(d)$ holds.

The considered constraining facets for deriving a datatype d are reviewed in Appendix A and are *length, minLength, maxLength, pattern, enumeration, whiteSpace, maxInclusive, minInclusive, maxExclusive, minExclusive, totalDigits*, and *fractionDigits*. The values of these facets should satisfy a number of constraints, presented in Appendix B (see also [23]). In Appendix C, we define the built-in XSD datatypes d that are derived from a primitive XSD datatype d', i.e. $primAncestor(d) = d'$ holds. These include the datatypes *integer, long, int, short, byte*, etc.

4 Valid RDF KBs and Datatype Evolution

Let K be a KB, where blank nodes are handled by skolemization[11]. The *D-closure* (or *D*-closure*) of a KB K, denoted by $\mathcal{C}(K)$, contains all the RDF triples of the form $(s\ p\ o)$, where $s,\ p \in URI$ and $o \in URI \cup LIT$, that either are explicitly asserted or can be entailed from K based on D-entailment of the RDF semantics [11] (or D^*-entailment, defined in [27]).

Now, we distinguish out of the set of properties of a KB, the set of XSD-valued properties. We denote the set of ranges of a property pr in a KB K by $range_K(pr)$.

Def. 1 (XSD-valued Property) Let K be a KB and let $pr \in Pr_K$. We say that pr is an *XSD-valued property* if there is $d \in range_K(pr)$ s.t. $D(d) \in XSD$. We denote the set of XSD-valued properties by Pr_K^{XSD}. □

We consider a KB K to be *valid* if all of the following conditions are satisfied:

(i) K has a D-interpretation (or D^*-interpretation), according to RDF semantics [11] (or [27]).
(ii) If $pr \in Pr_K^{XSD}$ then there are not $d, d' \in range_K(pr)$ s.t. $d \neq d'$ and $D(d), D(d') \in XSD$. We denote the unique $d \in range_K(pr)$ s.t. $D(d) \in XSD$, by $ran_K(pr)$.
(iii) If $pr \in Pr_K^{XSD}$ and $(o\ pr\ s\hat{}\hat{}u) \in K$ then $u = ran_K(pr)$.

[11] As have been shown in [5], D-entailment and D^*-entailment is not affected by replacing blank nodes by globally unique URIs.

(iv) If $pr \in Pr_K^{XSD}$, $(o\ pr\ s_1\hat{}\hat{}ran_K(pr)) \in K$, and $(o\ pr\ s_2\hat{}\hat{}ran_K(pr)) \in K$ then $V^d(s_1) \neq V^d(s_2)$, for $d = D(ran_K(pr))$.

Condition (iv), in the definition of a valid KB K, indicates that if $(o\ pr\ s_1\hat{}\hat{}ran_K(pr))$, $(o\ pr\ s_2\hat{}\hat{}ran_K(pr)) \in K$ then s_1 and s_2 should not map to the same value in the value space of $ran_K(pr)$. If $pr \in Pr_K^{XSD}$ and $(o\ pr\ s_1\hat{}\hat{}ran_K(pr)) \in K$ then, due to condition (i) in the definition of a valid KB K, $s_1\hat{}\hat{}ran_K(pr)$ is a well-typed literal.

Convention: In this work, we consider only valid KBs.

The following lemma is due to the fact that each KB K has a D-interpretation or a D^*-interpretation.

Lemma 1 Let K be a KB and $pr \in Pr_K^{XSD}$. If $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$ then $(o\ pr\ L2C^d(s)\hat{}\hat{}ran_K(pr)) \in \mathcal{C}(K)$, where $d = D(ran_K(pr))$. □

Below, we define the datatype direct evolution between two primitive XSD datatypes.

Def. 2 (Datatype Direct Evolution) We define the relation $\rightarrow$ between XSD datatypes, called *direct evolution*, as follows: *float* $\rightarrow$ *double*, *gYear* $\rightarrow$ *gYearMonth*, *gYearMonth* $\rightarrow$ *date*, *date* $\rightarrow$ *dateTime*, *gMonth* $\rightarrow$ *gMonthDay*, *hexBinary* $\rightarrow$ *base64Binary*, and *base64Binary* $\rightarrow$ *hexBinary*. □

We denote by $\rightarrow^*$, the transitive closure of the relation $\rightarrow$ between primitive XSD datatypes.

The following two definitions show how a value in the value space of a datatype d can be mapped to a value in the value space of a datatype d', if $primAncestror(d) = primAncestror(d')$ or $primAncestror(d) \rightarrow^* primAncestror(d')$. In fact, we provide two such mappings that in certain cases coincide.

Def. 3 (Literal Value Minimum Mapping) Let $d, d' \in XSD$ and let $x \in V^d$.[12] Let $d_1 = primAncestor(d)$ and $d_2 = primAncestor(d')$. We define $m(x, d, d')$ as follows:

- if $d_1 = d_2$ and $x \in V^{d'}$ then $m(x, d, d') = x$,
- $m(x, float, double) = x$,
- $m(x, gYear, gYearMonth)$ is the first month of the year $gYear$,
- $m(x, gYearMonth, date)$ is the first day of the month and year $gYearMonth$,
- $m(x, date, dateTime)$ is the first moment of the date $date$,
- $m(x, gMonth, gMonthDay)$ is the first date of the month $gMonth$,
- $m(x, gYear, date)$ is $m(m(x, gYear, gYearMonth), gYearMonth, date)$,
- $m(x, gYear, dateTime)$ is $m(m(x, gYear, date), date, dateTime)$,
- $m(x, gYearMonth, dateTime)$ is $m(m(x, gYearMonth, date), date, dateTime)$,
- $m(x, hexBinary, base64Binary)$ is x,
- $m(x, base64Binary, hexBinary)$ is x,
- if $d_1 \rightarrow^* d_2$, and $m(x, d_1, d_2) \in V^{d'}$ then $m(x, d, d') = m(x, d_1, d_2)$,
- in all other cases, $m(x, d, d')$ is undefined. □

Def. 4 (Literal Value Maximum Mapping) Let $d, d' \in XSD$ and let $x \in V^d$. Let $d_1 = primAncestor(d)$ and $d_2 = primAncestor(d')$. We define $M(x, d, d')$ as follows:

- if $d_1 = d_2$ and $x \in V^{d'}$ then $M(x, d, d') = m(x, d, d') = x$,

[12] Note that x is a value in the value space of d.

- $M(x, float, double) = x$,
- $M(x, gYear, gYearMonth)$ is the last month of the year $gYear$,
- $M(x, gYearMonth, date)$ is the last day of the month and year $gYearMonth$,
- $M(x, date, dateTime)$ is the last moment of the date $date$,
- $M(x, gMonth, gMonthDay)$ is the last date of the month $gMonth$,
- $M(x, gYear, date)$ is $M(M(x, gYear, gYearMonth), gYearMonth, date)$,
- $M(x, gYear, dateTime)$ is $M(M(x, gYear, date), date, dateTime)$,
- $M(x, gYearMonth, dateTime)$ is $M(M(x, gYearMonth, date), date, dateTime)$,
- $M(x, hexBinary, base64Binary) = m(x, hexBinary, base64Binary) = x$,
- $M(x, base64Binary, hexBinary) = m(x, base64Binary, hexBinary) = x$,
- if $d_1 \rightarrow^* d_2$, and $M(x, d_1, d_2) \in V^{d'}$ then $M(x, d, d') = M(x, d_1, d_2)$,
- in all other cases, $M(x, d, d')$ is undefined. □

Based on the functions $m(.)$ and $M(.)$, we define two other useful functions $n(s, d, d')$ and $N(s, d, d')$, where d, d' are datatypes and $s \in L^d$. In particular, $n(s, d, d') = C^{d'}(m(V^d(s), d, d'))$ and $N(s, d, d') = C^{d'}(M(V^d(s), d, d'))$. Note that $n(s, d, d') \in C^{d'}$, if $m(V^d(s), d, d')$ is defined, and undefined, otherwise. Similarly, $N(s, d, d') \in C^{d'}$, if $M(V^d(s), d, d')$ is defined, and undefined, otherwise.

Lemma 2 Let $d_1, d_2 \in XSD$ and let $s \in L^{d_1} \cap L^{d_2}$. Then, the following hold:
1. If $d_1 = d_2 = d$ then $n(s, d, d) = L2C^d(s)$.
2. If $primAncestor(d_1) = primAncestor(d_2) = d$ then $n(s, d_1, d_2) = N(s, d_1, d_2) = L2C^d(s) = L2C^{d_1}(s) = L2C^{d_2}(s)$. □

Having defined direct datatype evolution between primitive XSD datatypes, we now define datatype evolution between any XSD datatypes. This is actually, the kind of datatype evolution that we support.

Def. 5 (Datatype Evolution) Let $d, d' \in XSD$. We say that d *can evolve to* d', denoted by $d \Rightarrow d'$, if for all $x \in V^d$, it holds that $m(x, d, d') \in V^{d'}$, $M(x, d, d') \in V^{d'}$, and (i) $primAncestor(d) = primAncestor(d')$, or (ii) $primAncestor(d) \rightarrow^* primAncestor(d')$. □

Obviously, if $d \rightarrow^* d'$ then $d \Rightarrow d'$. Additionally, let $d_1, d_2, d_3 \in XSD$ s.t. $d_1 \Rightarrow d_2$ and $d_2 \Rightarrow d_3$ and let $x \in V^{d_1}$. Then, it is easy to see that (i) $m(m(x, d_1, d_2), d_3) = m(x, d_1, d_3)$, (ii) $M(M(x, d_1, d_2), d_3) = M(x, d_1, d_3)$, (iii) $n(n(x, d_1, d_2), d_3) = n(x, d_1, d_3)$, and (iv) $N(N(x, d_1, d_2), d_3) = N(x, d_1, d_3)$.

Example 2. Consider a datatype $d \in derFrom(integer)$ s.t. $d.minInclusive =$ "1" and $d.maxInclusive =$ "5". Additionally, consider a datatype $d' \in derFrom(decimal)$ s.t. $d'.minExclusive =$ "0" and $d'.maxInclusive =$ "7". It holds that $d \Rightarrow d'$. Now, consider a datatype $d \in derFrom(gYearMonth)$ s.t. $d.minInclusive =$ "1991 − 01" and $d.maxInclusive =$ "1993 − 12". Additionally, consider a datatype $d' \in derFrom(date)$ s.t. $d'.minExclusive =$ "1900 − 01 − 01" and $d'.maxInclusive =$ "2000 − 02 − 12". It holds that $d \Rightarrow d'$. □

Lemma 3 Let $d, d' \in XSD$ s.t. $d \Rightarrow d'$. Then, $ordered(d)$ holds iff $ordered(d')$ holds. □

The following proposition shows that the datatype evolution relationship $\Rightarrow$ is transitive.

Prop. 1 Let $d_1, d_2, d_3 \in XSD$ s.t. $d_1 \Rightarrow d_2$ and $d_2 \Rightarrow d_3$. Then, it holds that $d_1 \Rightarrow d_3$. □

5 Datatype Backwards Compatible Schema Evolution and Datatype Instance Tuples

In this section, we define datatype backwards compatible KB evolution and the set of datatype instance tuples, which we divide into the set of *possible* and *false* datatype instance tuples.

First, we define the XSD-valued instance triples of a KB K.

Def. 6 (XSD-valued Instance Triples) Let K be a KB. We define the *XSD-valued instance triples* of K, denoted by I_K^{XSD}, as follows:
$I_K^{XSD} = \{(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K \mid pr \in Pr_K^{XSD}\}$. □

Now , we define when a new KB K' is datatype backwards compatible with an old KB K. This is the kind of KB evolution that we support, called *datatype backwards compatible KB evolution*.

Def. 7 (Datatype Backwards Compatible) Let K, K' be two KBs. We say that K' is *datatype backwards compatible* with K, denoted by $K \gg K'$, if it holds that:

1. for all $pr \in Pr_K^{XSD}$, it holds that $ran_K(pr) \Rightarrow ran_{K'}(pr)$, and
2. $$I_{K'}^{XSD} = \{(o\ pr\ n(s,d,d')\hat{}\hat{}ran_{K'}(pr)) \mid (o\ pr\ s\hat{}\hat{}ran_K(pr)) \in I_K^{XSD}, \\ d = D(ran_K(pr)),\ \text{and}\ d' = D(ran_{K'}(pr))\} \cup \\ \{(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K' \mid pr \in Pr_{K'}^{XSD} \setminus Pr_K^{XSD}\}$$ □

Let K be a KB and K' be a new version of K s.t. $K \gg K'$. Additionally, let $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in I_K^{XSD}$. We assume that the RDF triple $(o\ pr\ n(s,d,d')\hat{}\hat{}ran_{K'}(pr)) \in I_{K'}^{XSD}$ has been generated automatically, by the system. Note that $n(s,d,d')\hat{}\hat{}ran_{K'}(pr)$ is a well-typed literal. This is the way that the XSD-valued instance triples of K are migrated to K'.

Example 3. Let K, K' be KBs s.t. $K \gg K'$. Additionally, let $pr \in Pr_K^{XSD}$ with $ran_K(pr) = xsd{:}integer$, $ran_{K'}(pr) = xsd{:}decimal$, and $(o\ pr\ \text{“}+1\text{”}\hat{}\hat{}xsd{:}integer) \in I_K^{XSD}$. Then, $(o\ pr\ \text{“}1\text{”}\hat{}\hat{}xsd{:}decimal) \in I_{K'}^{XSD}$. Further, let $pr \in Pr_K^{XSD}$ with $ran_K(pr) = xsd{:}gYear$, $ran_{K'}(pr) = xsd{:}date$, and $(o\ pr\ \text{“}2011\text{”}\hat{}\hat{}xsd{:}gYear) \in I_K^{XSD}$. Then, $(o\ pr\ \text{“}2011-01-01\text{”}\hat{}\hat{}xsd{:}date) \in I_{K'}^{XSD}$. □

The following proposition shows that the $\gg$ relationship is transitive.

Prop. 2 Let K_1, K_2, K_3 be KBs s.t. $K_1 \gg K_2$ and $K_2 \gg K_3$. Then, it holds that $K_1 \gg K_3$. □

Below, we define the *datatype instance tuples* of a KB K.

Def. 8 (Datatype Instance Tuples) Let K be a KB and let $pr \in Pr_K^{XSD}$. Additionally, let $d = D(ran_K(pr))$. We say that $(o,\ pr,\ s,\ s')$ is a *datatype instance tuple* of K, if there is $(o\ pr\ s_1\hat{}\hat{}ran_K(pr)) \in K$, $s = L2C^d(s_1)$, and $s' \in C^d$. We denote the set of datatype instance tuples by DIT_K. □

Prop. 3 Let K and K' be two KBs s.t. $K \gg K'$. Let $pr \in Pr_K^{XSD}$ s.t. $d = D(ran_K(pr))$ and $d' = D(ran_{K'}(pr))$.

1. If $d \Rightarrow d'$ and $(o,\ pr,\ s,\ s') \in DIT_K$ then $(o,\ pr,\ n(s,d,d'),\ n(s',d,d')) \in DIT_{K'}$.

2. If $d = d'$ and $(o,\ pr,\ s,\ s') \in DIT_K$ then $(o,\ pr,\ s,\ s') \in DIT_{K'}$. □

In the following, we shall partition DIT_K into two disjoint sets, *false* (denoted by F_K) and *possible* (denoted by P_K), i.e. $DIT_K = F_K \cup P_K$. We refer to this partition, as *D-partition*. Intuitively, P_K contains datatype instance tuples $(o,\ pr,\ s,\ s')$ s.t. $(o\ pr\ s'\hat{}\hat{}ran_K(pr))$ *can* possibly replace $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in \mathcal{C}(K)$, as a more "accurate" RDF triple, after the curation process (see Section 7). In contrast, F_K contains datatype instance tuples $(o,\ pr,\ s,\ s')$ s.t. $(o\ pr\ s'\hat{}\hat{}ran_K(pr))$ *cannot* possibly replace $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in \mathcal{C}(K)$, as a more "accurate" RDF triple, after the curation process.

Note that, due to Lemma 1, if $(o,\ pr,\ s,\ s') \in DIT_K$ then $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in \mathcal{C}(K)$. Moreover, if $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in I_K^{XSD}$ then $(o,\ pr,\ L2C^d(s),\ L2C^d(s)) \in P_K$, where $d = D(ran_K(pr))$.

5.1 Formalizing *MLSA* through *D*-partition

Let K be a KB. We say that an RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in I_K^{XSD}$ satisfies the *Maximum Literal Specificity Assumption* (*MLSA*), if $s\hat{}\hat{}ran_K(pr)$ is the most accurate literal value for property pr and subject o. Further, we say that K satisfies the *MLSA*, if for all $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in I_K^{XSD}$, it holds that $(o\ pr\ s\hat{}\hat{}ran_K(pr))$ satisfies the *MLSA*.

Obviously, if K satisfies the *MLSA* then, for all $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in I_K^{XSD}$, it holds that $(o,\ pr,\ L2C^d(s),\ L2C^d(s)) \in P_K$, where $d = D(ran_K(pr))$, and P_K contains nothing else. That is, $P_K = \{(o, pr, s, s') \in DIT_K \mid s = s'\}$. Now, it holds that $F_K = DIT_K \setminus P_K = \{(o, pr, s, s') \in DIT_K \mid s \neq s'\}$.

Initially, a KB K satisfies the *MLSA*. However, as we move from a KB K to a KB K' s.t. $K \gg K'$ (denoted by $K \rightsquigarrow K'$), the *MLSA* may not be satisfied. For example, let $pr \in Pr_K^{XSD}$ with $ran_K(pr) = xsd{:}integer$, $ran_{K'}(pr) = xsd{:}decimal$, and $(o\ pr\ \text{“}+1\text{”}\hat{}\hat{}xsd{:}integer) \in I_K^{XSD}$. Then, $(o\ pr\ \text{“}1\text{”}\hat{}\hat{}xsd{:}decimal) \in I_{K'}^{XSD}$. Though, $\text{“}+1\text{”}\hat{}\hat{}xsd{:}integer$ is the most accurate literal value for property pr and subject o in K. The same is not true for $\text{“}1\text{”}\hat{}\hat{}xsd{:}decimal$ in K' and the most accurate literal value for property pr and subject o in K' may be $\text{“}x\text{”}\hat{}\hat{}xsd{:}decimal$, for $x \in (0, 2)$. Let us say that the most accurate literal value for property pr and subject o in K' is $\text{“}0.7\text{”}\hat{}\hat{}xsd{:}decimal$. Then, after the curation process, the system can replace the RDF triple $(o\ pr\ \text{“}1\text{”}\hat{}\hat{}xsd{:}decimal)$ in K' by $(o\ pr\ \text{“}0.7\text{”}\hat{}\hat{}xsd{:}decimal)$. Now, the RDF triple $(o\ pr\ \text{“}0.7\text{”}\hat{}\hat{}xsd{:}decimal) \in K'$ satisfies the *MLSA*.

Thus, let K, K' be KBs s.t. $K \rightsquigarrow K'$. Our objective is to define the new *D*-partition, i.e. we want to define the transition from (F_K, P_K) to $(F_{K'}, P_{K'})$, denoted by $(F_K, P_K) \rightsquigarrow (F_{K'}, P_{K'})$. By moving from K to K', the rising question is how $F_{K'}$ can be defined, based on F_K (note that $P_{K'} = DIT_{K'} \setminus F_{K'}$).

The following definition shows how a datatype instance tuple in F_K, affects the definition of $F_{K'}$.

Def. 9 Let K, K' be KBs s.t. $K \rightsquigarrow K'$. Let $(o,\ pr,\ s,\ s') \in F_K$, $d = D(ran_K(pr))$, and $d' = D(ran_{K'}(pr))$[13].

*R*1: If $ordered(d)$ holds and $V^d(s') < V^d(s)$ then for all $v \in V^{d'}$ s.t. $v \leq M(V^d(s'), d, d')$, it holds that $(o,\ pr,\ n(s, d, d'),\ C^{d'}(v)) \in F_{K'}$.

[13] Note that only the values in the value space of a datatype can be compared.

*R*2: If $ordered(d)$ holds and $V^d(s') > V^d(s)$ then for all $v \in V^{d'}$ s.t. $v \geq m(V^d(s'), d, d')$, it holds that $(o,\ pr,\ n(s,d,d'),\ C^{d'}(v)) \in F_{K'}$.
*R*3: If $ordered(d)$ does not hold then $(o,\ pr,\ n(s,d,d'),\ n(s',d,d')) \in F_{K'}$.
□

Let K, K' be KBs s.t. $K \rightsquigarrow K'$. If $pr \in Pr^{XSD}_{K'} \setminus Pr^{XSD}_{K}$ and $t = (o\ pr\ \text{“}s\text{”}\hat{}\hat{}\ ran_{K'}(pr)) \in I^{XSD}_{K'}$ then we assume that t satisfies the *MLSA*. That is, $F_{K'}$ contains the tuples $(o, pr, L2C^d(s), s') \in DIT_{K'}$, where $d = D(ran_{K'}(pr))$ and $s' \neq L2C^d(s)$. This is reasonable, since pr is a new XSD-valued property of K'.

Example 4. Consider a KB K for which the *MLSA* holds. Assume that the range of property $pr \in Pr^{XSD}_{K}$ is *xsd*:*integer* and $(o\ pr\ \text{“}+1\text{”}\hat{}\hat{}xsd{:}integer) \in K$. Note that $L2C^{integer}(\text{“}+1\text{”}) = \text{“1”}$. Then, $(o, pr, \text{“1”},\ C^{integer}(v)) \in F_K$, for all $v \in V^{integer}$ s.t. $v \leq 0$ or $v \geq 2$. Consider now a KB K' s.t. $K \rightsquigarrow K'$. Assume that the range of pr in K' is *xsd*:*decimal*. Note that $n(\text{“1”}, integer, decimal) = \text{“1”}$, $M(0, integer, decimal) = 0$, and $m(2, integer, decimal) = 2$. Therefore, using rules *R*1 and *R*2, $(o, pr, \text{“1”},\ C^{decimal}(v)) \in F_{K'}$, for all $v \in V^{decimal}$ s.t. $v \leq 0$ or $v \geq 2$. Additionally, $(o, pr, \text{“1”},\ C^{decimal}(v)) \in P_{K'}$, for all $v \in V^{decimal}$ s.t. $0 < v < 2$. This means that the RDF triple $(o\ pr\ \text{“1”}\hat{}\hat{}xsd{:}decimal) \in K'$ can be possibly replaced (after the curation process) by any RDF triple $(o\ pr\ \ s\hat{}\hat{}xsd{:}decimal)$, where $s \in L^{decimal}$ s.t. $0 < V^{decimal}(s) < 2$. □

Example 5. Consider a KB K and a property $pr \in Pr^{XSD}_{K}$ with range *ex*:*two_colors*, where *ex*:*two_colors* has been derived from *xsd*:*string* through the enumeration of the values “white” and “black”. Consider now a KB K' s.t. $K \rightsquigarrow K'$. Assume that the range of pr in K' is *ex*:*three_colors*, where *ex*:*three_colors* has been derived from *xsd*:*string* through the enumeration of the values “white”, “black”, and “grey”. Obviously, *two_colours* ⇒ *three_colours* and $ordered(two\_colours)$ does not hold. Let $(o,\ pr,\ \text{“}white\text{”},\ \text{“}black\text{”}) \in F_K$. Note that $n(\text{“}white\text{”}, two\_colours, three\_colours) = \text{“}white\text{”}$ and $n(\text{“}black\text{”}, two\_colours, three\_colours) = \text{“}black\text{”}$. Thus, using Rule *R*3, $(o,\ pr,\ \text{“}white\text{”},\ \text{“}black\text{”}) \in F_{K'}$, while $(o,\ pr,\ \text{“}white\text{”},\ \text{“}grey\text{”}) \in P_{K'}$. Therefore, the RDF triple $(o\ pr\ \ \text{“}grey\text{”}\hat{}\hat{}\ ex{:}three\_colors)$ can possibly replace the RDF triple $(o\ pr\ \ \text{“}white\text{”}\hat{}\hat{}ex{:}three\_colors) \in K'$, after the curation process. □

The following propositions show interesting properties of F_K and P_K.

Prop. 4 Let $(o, pr, s, s') \in F_K$ s.t. $ordered(d)$ holds, for $d = D(ran_K(pr))$. It holds that:
1. If $V^d(s') < V^d(s)$ then, for all $s'' \in C^d$ s.t. $V^d(s'') \leq V^d(s')$, $(o, pr, s, s'') \in F_K$.
2. If $V^d(s') > V^d(s)$ then, for all $s'' \in C^d$ s.t. $V^d(s'') \geq V^d(s')$, $(o, pr, s, s'') \in F_K$.
□

Prop. 5 Let $(o, pr, s, s_1), (o, pr, s, s_3) \in P_K$ s.t. $ordered(d)$ holds, for $d = D(ran_K(pr))$. If $V^d(s_1) < V^d(s_3)$ then, for all $s_2 \in C^d$ s.t. $V^d(s_1) \leq V^d(s_2) \leq V^d(s_3)$, $(o, pr, s, s_2) \in P_K$. □

The following proposition relates P_K with $P_{K'}$.

Prop. 6 Let K, K' be KBs s.t. $K \rightsquigarrow K'$. If $(o, pr, s, s') \in P_K$ then $(o, pr, n(s, d, d'), n(s', d, d')) \in P_{K'}$, where $d = D(ran_K(p))$ and $d' = D(ran_{K'}(pr))$. □

The following proposition indicates that the possible and false datatype instance tuples derived based on successive KB version transitions are the same as the possible and false datatype instance tuples derived based on the direct KB transition from the first version to the last version, provided that the set of XSD-valued properties of the different KB versions is the same.

Prop. 7 Let K_1, K_2, K_3 be KBs s.t. $K_1 \rightsquigarrow K_2 \rightsquigarrow K_3$ and $Pr^{XSD}_{K_1} = Pr^{XSD}_{K_2}$. Consider the one step transition $(F_{K_1}, P_{K_1}) \rightsquigarrow (F_{K_3}, P_{K_3})$ and the successive step transitions $(F_{K_1}, P_{K_1}) \rightsquigarrow (F'_{K_2}, P'_{K_2}) \rightsquigarrow (F'_{K_3}, P'_{K_3})$. It holds that $P_{K_3} = P'_{K_3}$ and $F_{K_3} = F'_{K_3}$. □

However, the result of Prop. 7 may not hold in the case that $Pr^{XSD}_{K_1} \subset Pr^{XSD}_{K_2}$, as shown in Appendix F.

6 Deriving the Set of Possible Datatype Instance Tuples

Let K, K' be KBs s.t. $K \rightsquigarrow K'$. In the previous section, we described how $F_{K'}$ is generated from F_K. Now, the rising question is how we compute $P_{K'}$ based on P_K, since it is more user-friendly to present to the user positive possible information than negative information. However, $P_{K'}$, as well as $DIT_{K'}$ and $F_{K'}$, can be possibly infinite sets. Therefore, we need a finite representation of P_K, denoted by $P^{\mathtt{fin}}_K$. In this section, we define $P^{\mathtt{fin}}_K$ and we describe how $P^{\mathtt{fin}}_K$ is derived if K satisfies the *MLSA* and how $P^{\mathtt{fin}}_{K'}$ is derived based on $P^{\mathtt{fin}}_K$.

6.1 Finite Representation of Possible Datatype Instance Tuples

We define a finite set $P^{\mathtt{fin}}_K$, where each tuple $t \in P^{\mathtt{fin}}_K$ represents a set of possible datatype instance tuples of K, denoted by $PossRep_K(t)$. It holds that (i) $P_K = \cup\{PossRep_K(t) \mid t \in P^{\mathtt{fin}}_K\}$, (ii) for all $t, t' \in P^{\mathtt{fin}}_K$ s.t. $PossRep_K(t) \cap PossRep_K(t') = \emptyset$, and (iii) for all $t \in P^{\mathtt{fin}}_K$, whose first three arguments are o, pr, s then $(o, pr, s, s) \in PossRep_K(t)$. In particular (let $order_1, order_2, order \in \{<, \leq\}$):

- if $(o,\ pr,\ s,\ order_1\ s_1,\ order_2\ s_2) \in P^{\mathtt{fin}}_K$ then, for $d = D(ran_K(pr))$, it holds that $s_1, s_2 \in C^d$ and $PossRep_K(t) = \{(o,\ pr,\ s,\ s') \in DIT_K \mid V^d(s_1)\ order_1\ V^d(s')\ order_2\ V^d(s_2)\}$,[14]
- if $(o,\ pr,\ s,\ \text{“}\geq\text{LOW”},\ order\ s_2) \in P^{\mathtt{fin}}_K$ then, for $d = D(ran_K(pr))$, it holds that $s_2 \in C^d$ and $PossRep_K(t) = \{(o,\ pr,\ s,\ s') \in DIT_K \mid min(V^d) \leq V^d(s')\ order\ V^d(s_2)\}$,
- if $(o,\ pr,\ s,\ order\ s1,\ \text{“}\leq\text{HIGH”}) \in P^{\mathtt{fin}}_K$ then, for $d = D(ran_K(pr))$, it holds that $s_1 \in C^d$ and $PossRep_K(t)) = \{(o,\ pr,\ s,\ s') \in DIT_K \mid V^d(s_1)\ order\ V^d(s') \leq max(V^d)\}$,
- if $(o,\ pr,\ s,\ \text{“}\geq \text{LOW”},\ \text{“}\leq \text{HIGH”}) \in P^{\mathtt{fin}}_K$ then, for $d = D(ran_K(pr))$, it holds that $PossRep_K(t) = \{(o,\ pr,\ s,\ s') \in DIT_K \mid min(V^d) \leq V^d(s') \leq max(V^d)\}$,

[14] Recall that only the values in the value space of a datatype can be compared.

– if $(o,\ pr,\ s,\ (s_1, u)) \in P_K^{\mathtt{fin}}$ then, for $d = D(ran_K(pr))$, (i) $ordered(d)$ does not hold, (ii) u is a datatype s.t. $u \Rightarrow d$, and (iii) $s_1 \in C^u$. Additionally, $PossRep_K(t) = \{(o,\ pr,\ s,\ s') \in DIT_K \mid s' \in (C^d \setminus n(L^{D(u)}, D(u), d)) \cup \{n(s_1, D(u), d)\}\}$.[15] Note that s' is either a canonical value of d, that is not the mapping of a lexical value of $D(u)$ to d, or s' is the mapping of s_1 to d.

We refer to the instances of $P_K^{\mathtt{fin}}$, as *compact possible datatype tuples.*[16] Additionally, if $t = (o, pr, s, s_1, s_2) \in P_K^{\mathtt{fin}}$ then we say that t *corresponds to* the RDF triple $(o\ pr\ s''\hat{}\hat{}ran_K(pr)) \in I_K^{XSD}$, where $s = L2C^{D(ran_K(pr))}(s'')$.

Lemma 4 Let K be a KB. The following hold:
1. For each $t \in I_K^{XSD}$, there is a unique $t' \in P_K^{\mathtt{fin}}$ that corresponds to t.
2. For each $t \in P_K^{\mathtt{fin}}$, there is a unique $t' \in I_K^{XSD}$ s.t. t corresponds to t'.
3. $|P_K^{\mathtt{fin}}| = |I_K^{XSD}|$. □

Obviously, an RDF triple $t = (o\ pr\ s\hat{}\hat{}ran_K(pr)) \in I_K^{XSD}$ satisfies the *MLSA* if $PossRep_K(t') = \{(o, pr, L2C^d(s), L2C^d(s))\}$, for the tuple $t' \in P_K^{\mathtt{fin}}$ that corresponds t, where $d = D(ran_K(pr))$.

Example 6. Consider an KB K and a property $pr \in Pr_K^{XSD}$ with $ran_K(pr) = xsd{:}decimal$. Assume now that $t = (o, pr, \text{“2”}, \text{“} > 1\text{”}, \text{“} < 3\text{”}) \in P_K^{\mathtt{fin}}$. Then, $PossRep_K(t) = \{(o, pr, \text{“2”}, s')\}$, where $s' \in C^{decimal}$ and $1 < V^{decimal}(s') < 3$. Thus, t represents an infinite number of possible datatype instance tuples. □

We refer to a tuple $t \in P_K^{\mathtt{fin}}$ s.t. $PossRep_K(t) = \{(o, pr, L2C^d(s), L2C^d(s))\}$, where $d = D(ran_K(pr))$, as *MLSA tuple*.

6.2 Deriving the Original $P_K^{\mathtt{fin}}$ Set

Let K be a KB for which the *MLSA* holds. Our goal is to identify for each $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in I_K^{XSD}$, the tuple that needs to be added to $P_K^{\mathtt{fin}}$. To achieve this we have to consider the values of the declared facets for each datatype d' starting from $d = D(ran_K(pr))$ up to $primAncestor(d)$. We consider that for each datatype d and constraining facet F there is a field $d.F$. If F is a declared facet of d with value f then $d.F = f$ and if F is not a declared facet of d then, initially, $d.F$ =NULL. Algorithm *IdentifyDatatypeMin*(d) (resp. *IdentifyDatatypeMax*(d)) takes as input a datatype d s.t. $ordered(d)$ holds and identifies the values of $d.minInclusive$ or $d.minExclusive$ (resp. $d.maxInclusive$ or $d.maxExclusive$) that are either declared or inherited from the ancestor datatypes. Algorithm *IdentifyTotalDigits*(d) (resp. *IdentifyFractionDigits*(d)) takes as input a datatype d s.t. $primAncestor(d) = decimal$ and identifies the value of $d.totalDigits$ (resp. $d.fractionDigits$) that is either declared or inherited from the ancestor datatypes. These algorithms are presented in Appendix D.

Below, we present Algorithm 1, *InitDatatypePossibilities*(K), which given a KB K for which the *MLSA* holds, it returns that initial set $P_K^{\mathtt{fin}}$. The tuples in this initial set $P_K^{\mathtt{fin}}$ are set in such a way that the derivation of $P_{K'}^{\mathtt{fin}}$, when K evolves to another datatype backwards compatible KB K', becomes straightforward (see Algorithm 4).

---

[15] Let d, d' be datatypes and $S \subseteq L^d$. We define $n(S, d, d') = \{n(s, d, d') \mid s \in S\}$.

[16] Note that $P_K^{\mathtt{fin}}$ is a set of 5-tuples, where each 5-tuple t represents a (possibly infinite) set of 4-tuples, through that function $PossRep_K(t)$.

Algorithm 1 *InitDatatypePossibilities*(K)
Input: A KB K for which the *MLSA* holds
Output: The initial set $P_K^{\mathtt{fin}}$

(1) $P_K^{\mathtt{fin}} := \emptyset$;
(2) For all $pr \in Pr_K^{XSD}$ do
(3) $\quad d := D(ran_K(pr))$;
(4) $\quad$ For all $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$ do
(5) $\quad\quad P_K^{\mathtt{fin}} = P_K^{\mathtt{fin}} \cup$ *InitDatatype*$(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$;
(6) Return$(P_K^{\mathtt{fin}})$;

Algorithm 1, *InitDatatypePossibilities*(K), initially sets $P_K^{\mathtt{fin}}$ to empty set. Then, for each $pr \in Pr_K^{XSD}$, it sets $d = D(ran_K(pr))$. Now, for each RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$, it calls Algorithm *InitDatatype*$(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$, which returns a tuple t to be added to the initial $P_K^{\mathtt{fin}}$. Finally, it returns the initial set $P_K^{\mathtt{fin}}$.

Below, we present Algorithm 2, *InitDatatype*$(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$, that given a KB K and an RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$, for which the *MLSA* holds, it returns an *MLSA* tuple t to be added to $P_K^{\mathtt{fin}}$.

Algorithm 2 *InitDatatype*$(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$
Input: A KB K and an RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$, for which the *MLSA* holds
Output: An *MLSA* tuple t to be added to $P_K^{\mathtt{fin}}$

(1) $d := D(ran_K(pr))$;
(2) If $d.enumeration \neq$ NULL then
(3) $\quad$ Return(*InitPossEnumeration*$(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$);
(4) else
(5) $\quad$ If *ordered*(d) holds then
(6) $\quad\quad$ Let $Type = primAncestor(d)$;
(7) $\quad\quad$ Return(*InitPossType*$(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$);
(8) $\quad$ else /* *ordered*(d) does not hold */
(9) $\quad\quad$ Return$\{(o,\ pr,\ L2C^d(s),\ (L2C^d(s), ran_K(pr)))\}$;

Algorithm 2, *InitDatatype*$(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$, initially sets $d = D(ran_K(pr))$. Then, it checks the properties of d. In particular, if d is an enumerated datatype, it calls Algorithm 3, *InitPossEnumeration*$(K, (o\ pr\ \ s\hat{}\hat{}ran_K(pr)))$, which returns an *MLSA* tuple t to be added to $P_K^{\mathtt{fin}}$. Else, if *ordered*(d) holds, depending on the primitive XSD datatype that d is derived from, the appropriate algorithm is called which returns an *MLSA* tuple t to be added to $P_K^{\mathtt{fin}}$. Then, tuple t is returned by Algorithm 2. If, on the other hand, *ordered*(d) does not hold then the tuple $t = (o, pr, L2C^d(s), (L2C^d(s), ran_K(pr)))$ is returned by Algorithm 2 to be added to $P_K^{\mathtt{fin}}$. Note that $PossRep_K(t) = \{(o, pr, L2C^d(s), L2C^d(s))\}$. This is because $PossRep_K(t) = \{(o, pr, L2C^d(s), s_2) \mid s_2 \in (C^d \setminus n(L^d, d, d)) \cup \{n(L2C^d(s), d, d)\}\} = \{(o, pr, L2C^d(s), L2C^d(s))\}$[17].

Example 7. Consider a KB K and a property $hasName \in Pr_K^{XSD}$ s.t. $ran_K(hasName)$ $= ex{:}stringMax20$, which is derived from $xsd{:}string$ by setting the facet $maxLength$

[17] Note that $n(L^d, d, d) = C^d$ and $n(L2C^d(s), d, d) = L2C^d(s)$.

= 20. Let $d = string$. Note that $ordered(d)$ does not hold and $L2C^d(\text{“}Dom.\ Theotocopoulos\text{”}) = \text{“}Dom.\ Theotocopoulos\text{”}$. Then, $InitDatatype(K, (o,\ hasName\ \text{“}Dom.Theotocopoulos\text{”}\ \hat{}\hat{}ex{:}stringMax20))=(o, hasName, \text{“}Dom.Theotocopoulos\text{”}, (\text{“}Dom.\ Theotocopoulos\text{”}, ex{:}stringMax20))$. Note that $PossRep_K((o, hasName, \text{“}Dom.\ Theotocopoulos\text{”}, (\text{“}Dom.\ Theotocopoulos\text{”}, ex{:}stringMax20))) = \{(o, hasName, \text{“}Dom.\ Theotocopoulos\text{”}, \text{“}Dom.\ Theotocopoulos\text{”})\}$. □

Below, we present Algorithm 3, $InitPossEnumeration(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$, which takes as input a KB K and an RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$ for which the *MLSA* holds, where $pr \in Pr_K^{XSD}$ s.t. $D(ran_K(pr)).enumeration \neq$NULL. The algorithm returns an *MLSA* tuple to be added to $P_K^{\mathtt{fin}}$.

Algorithm 3 $InitPossEnumeration(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$
Input: A KB K and an RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$ for which the *MLSA* holds, where $pr \in Pr_K^{XSD}$ s.t. $D(ran_K(pr)).enumeration \neq$NULL
Output: An *MLSA* tuple to be added to $P_K^{\mathtt{fin}}$

```
( 1)  d := D(ran_K(pr));
( 2)  s' := L2C^d(s);
( 3)  If ordered(d) holds then
( 4)      If V^d(s) = min({V^d(s'') | s'' ∈ d.enumeration}) then
( 5)          s_1 := " ≥LOW";
( 6)      else
( 7)          v_1 := max({V^d(s'') | s'' ∈ d.enumeration and V^d(s'') < V^d(s)});
( 8)          s_1 := concatString(" > ", C^d(v_1));
( 9)      If V^d(s) = max({V^d(s'') | s'' ∈ d.enumeration}) then
(10)          s_2 := " ≤HIGH";
(11)      else
(12)          v_2 := min({V^d(s'') | s'' ∈ d.enumeration and V^d(s'') > V^d(s)});
(13)          s_2 := concatString(" < ", C^d(v_2));
(14)      Return((o, pr, s', s_1, s_2));
(15)  If ordered(d) does not hold then
(16)      Return((o, pr, s', (s', ran_K(pr))));
```

Initially, Algorithm $InitPossEnumeration(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$ sets $d = D(ran_K(pr))$ and $s' = L2C^d(s)$. Assume that $ordered(d)$ holds. If $V^d(s)$ is equal to the minimum value of the elements of $d.enumeration$ then s_1 is set equal to “ ≥LOW”. Otherwise, v_1 is set equal to the maximum value of the elements whose value is lower than $V^d(s)$. Then, $s_1 = concatString(\text{“ > ”}, C^d(v_1))$. Similarly, if $V^d(s)$ is equal to the maximum value of the elements of $d.enumeration$ then s_2 is set equal to “ ≤HIGH”. Otherwise, v_2 is set equal to the minimum value of the elements whose value is greater than $V^d(s)$. Then, $s_2 = concatString(\text{“ < ”}, C^d(v_2))$. Finally, the tuple $(o,\ pr,\ s',\ s_1,\ s_2)$ is returned to be added to $P_K^{\mathtt{fin}}$. If, on the other hand, $ordered(d)$ does not hold then the tuple $t = (o, pr, s', (s', ran_K(pr)))$ is returned to be added to $P_K^{\mathtt{fin}}$. This is because $PossRep_K(t) = \{(o,\ pr,\ s',\ s')\}$.

Example 8. Consider a KB K and a property $hasGrade \in Pr_K^{XSD}$ s.t. $ran_K($ $hasGrade) = ex{:}five\_integers$, which is derived from $xsd{:}integer$ through the enumeration of the five values “-1”, “0”, “2”, “3”, and “4”. Let $d = integer$. Note that $ordered(d)$ holds and $L2C^d(\text{“2”}) = \text{“2”}$. Then, $InitPossEnumeration(K, (o$

hasGrade "2"^^*ex:five_integers*))= $(o, hasGrade, \text{“2”}, \text{“} > 0\text{”}, \text{“} < 3\text{”})$. Note that $PossRep_K((o,\ hasGrade, \text{“2”}, \text{“} > 0\text{”}, \text{“} < 3\text{”})) = \{(o, hasGrade, \text{“2”}, \text{“2”})\}$. □

Below, we describe Algorithm $InitPossDecimal(K, (o\ pr\ s\hat{}\hat{}u))$, which takes as input a KB K and an RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$ for which the *MLSA* holds, where $pr \in Pr_K^{XSD}$ s.t. $D(ran_K(pr)).enumeration$ =NULL and $primAncestor(D(ran_K(pr))) = decimal$. The algorithm returns an *MLSA* tuple to be added to $P_K^{\mathtt{fin}}$ and is presented in Appendix D.

Initially, Alg. $InitPossDecimal(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$ sets $d = D(ran_K(pr))$ and calls the algorithms $IdentifyDatatypeMin(d)$, $IdentifyDatatypeMax(d)$, $IdentifyDatatypeTotalDigits(d)$, and $IdentifyDatatypeFractionDigits(d)$, assigning the returned value to d. Note that the call $IdentifyDatatypeFractionDigits(d)$ updates the field $d.fractionDigits$. Then, it sets $s' = L2C^d(s)$. Assume that $min(V^d)$ is defined. This means that (i) $d.minInclusive$ ≠NULL, or (ii) $d.totalDigits$ ≠ NULL, or (iii) $d.minExclusive$ ≠NULL and $d.fractionDigits$ ≠NULL. If $V^d(s)$ is the minimum value of V^d then s_1 is set “ ≥LOW”. Otherwise, if $d.totalDigits$ and $d.fractionDigits$ are not both equal to NULL then s_1 is set equal to $concatString($ “ > ”, $C^d(previousDecimal(\ V^d(s), d)))$[18]. If $d.totalDigits$ and $d.fractionDigits$ are both equal to NULL then s_1 is set equal to $concatString(\text{“} \geq \text{”}, s')$. Assume now that $min(V^d)$ is not defined. If $d.fractionDigits \neq$ NULL then s_1 is set equal to $concatString(\text{“} > \text{”}, C^d(previousDecimal(V^d(s), d)))$, else s_1 is set equal to $concatString(\text{“} \geq \text{”}, s')$. Similar is the case for defining s_2. Finally, the tuple $(o,\ pr,\ s',\ s_1,\ s_2)$ is returned to be added to $P_K^{\mathtt{fin}}$.

Example 9. Consider a KB K and a property $hasWeight \in Pr_K^{XSD}$ s.t. $ran_K(hasWeight)$ = *ex:decimalTotalDigits5*, which is derived from *xsd:decimal* by setting the facet $totalDigits = 5$. Let $d = decimal$. Note that $L2C^d(\text{“} + 253\text{”}) = \text{“253”}$. Then, $InitPossDecimal(K, (o\ hasWeight\ \text{“+253”}\hat{}\hat{}ex:decimalTotalDigits5))$=$(o, hasWeight, \text{“253”}, \text{“} > 252.99\text{”}, \text{“} < 253.01\text{”})$. Note that $PossRep_K((o, hasWeight, \text{“253”}, \text{“} > 252.99\text{”}, \text{“} < 253.01\text{”})) = \{(o, hasWeight, \text{“253”}, \text{“253”})\}$. Additionally, note that $L2C^d(\text{“}+99999\text{”}) = \text{“99999”}$ and that 99999 is the maximum decimal with five total digits. Then, $InitPossDecimal(K,(o\ hasWeight\ \text{“+99999”}\hat{}\hat{}\ ex:decimalTotalDigits5))$ =$(o, hasWeight, \text{“99999”}, \text{“} > 99998\text{”}, \text{“} \leq \text{HIGH”})$. Note that $PossRep_K((o, hasWeight, \text{“99999”}, \text{“} > 99998\text{”}, \text{“} \leq \text{HIGH”}) = \{(o, hasWeight, \text{“99999”}, \text{“99999”})\}$. □

For readability reasons, Algorithms $InitPossFloat(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$ and $InitPossDate(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$, called by Algorithm 2, $InitDatatype(\ K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$, are presented in Appendix D. Now, Algorithms $InitPossgYear(.)$, $InitPossgYearMonth(.)$, $InitPossgMonth(.)$, $InitPossgMonthDay(.)$, $InitPossgDay(.)$, $InitPossDateTime(.)$, and $InitPossTime(.)$ also called by Algorithm 2, are defined similarly to $InitPossDate(K,\ (o\ pr\ s\hat{}\hat{}\ ran_K(pr)))$.

The following proposition provides the time complexity of Algorithm 1, $InitDatatypePossibilities(K)$.

[18] Note that $previousDecimal(V^d(s), d)$ (Algorithm 10 in Appendix C) returns the largest decimal that is less than $V^d(s)$. Similarly, $nextDecimal(V^d(s), d)$ (Algorithm 11 in Appendix C) returns the smallest decimal that is greater than $V^d(s)$. Both algorithms are well defined if $d.totalDigits$ and $d.fractionDigits$ are not both equal to NULL.

Prop. 8 The time complexity of Algorithm *InitDatatypePossibilities*(K) is in $O(|I_K^{XSD}| * L)$, where L is the length of the longest datatype derivation chain for the XSD datatypes, considered in K. □

6.3 Deriving the Evolved $P_{K'}^{\mathtt{fin}}$ Set

Below, we present Algorithm 4, *DatatypePossibilities*$(K, K', P_K^{\mathtt{fin}})$, which takes as input two KBs K, K' s.t. $K \rightsquigarrow K'$ and the set $P_K^{\mathtt{fin}}$, and returns as output the new set $P_{K'}^{\mathtt{fin}}$ (for which it holds $P_{K'} = \cup\{PossRep_{K'}(t) \mid t \in P_{K'}^{\mathtt{fin}}\}$), i.e. the one corresponding to the transition $(F_K, P_K) \rightsquigarrow (F_{K'}, P_{K'})$.

Algorithm 4 *DatatypePossibilities*$(K, K', P_K^{\mathtt{fin}})$
Input: Two KBs K, K' s.t. $K \rightsquigarrow K'$ and the set $P_K^{\mathtt{fin}}$
Output: The set $P_{K'}^{\mathtt{fin}}$ [19]

(1) $P_{K'}^{\mathtt{fin}} := \emptyset$;
(2) For all $pr \in Pr_K^{XSD}$ do
(3) $\quad d := D(ran_K(pr))$;
(4) $\quad d' := D(ran_{K'}(pr))$;
(5) $\quad$ For all $(o,\ pr,\ s,\ order_1\ s_1,\ order_2\ s_2) \in P_K^{\mathtt{fin}}$ do
(6) $\quad$ /* Note that $order_1 \in \{>, \geq\}$ and $order_2 \in \{<, \leq\}$ */
(7) $\quad\quad$ If $primAncestor(d) = primAncestor(d')$ then
(8) $\quad\quad\quad P_{K'}^{\mathtt{fin}} := P_{K'}^{\mathtt{fin}} \cup \{(o,\ pr,\ s,\ order_1\ s_1,\ order_2\ s_2)\}$;
(9) $\quad\quad$ else
(10) $\quad\quad\quad P_{K'}^{\mathtt{fin}} := P_{K'}^{\mathtt{fin}} \cup \{(o,\ pr,\ n(s,d,d'),\ order_1\ N(s_1,d,d'),\ order_2\ n(s_2,d,d')\}$;
(11) $\quad$ For all $(o,\ pr,\ s,\ (s_1, u)) \in P_K^{\mathtt{fin}}$ do
(12) $\quad\quad s' := n(s,d,d')$;
(13) $\quad\quad P_{K'}^{\mathtt{fin}} := P_{K'}^{\mathtt{fin}} \cup \{(o,\ pr,\ s',\ (s_1, u))\}$;
(14) For all $pr \in Pr_{K'}^{XSD} \setminus Pr_K^{XSD}$ do
(15) $\quad$ For all $(o\ pr\ s\hat{}\hat{}ran_{K'}(pr)) \in K'$ do
(16) $\quad\quad P_{K'}^{\mathtt{fin}} := P_{K'}^{\mathtt{fin}} \cup \{InitDatatype(K', (o\ pr\ s\hat{}\hat{}ran_{K'}(pr))\}$;
(17) Return$(P_{K'}^{\mathtt{fin}})$;

Algorithm *DatatypePossibilities*$(K, K', P_K^{\mathtt{fin}})$, initially sets $P_{K'}^{\mathtt{fin}}$ to emptyset. Then, for all $pr \in Pr_K^{XSD}$, it sets $d = D(ran_K(pr))$ and $d' = D(ran_{K'}(pr))$. For all $(o,\ pr,\ s,\ order_1\ s_1,\ order_2\ s_2) \in P_K^{\mathtt{fin}}$, where $order_1 \in \{>, \leq\}$ and $order_2 \in \{<, \leq\}$, it does the following: (i) if $primAncestor(d) = primAncestor(d') = d''$ then it adds to $P_{K'}^{\mathtt{fin}}$ the tuple $(o,\ pr,\ s,\ order_1\ s_1,\ order_2\ s_2)$, due to rules $R1$ and $R2$ of Def. 9. Note that $N(s_1, d, d') = C^{d'}(M(V^d(s_1), d, d')) = C^{d'}(V^d(s_1)) = C^{d''}(V^d(s_1)) = C^d(V^d(s_1)) = s_1$ and $n(s_2, d, d') = C^{d'}(m(V^d(s_2), d, d')) = C^{d'}(V^d(s_2)) = C^{d''}(V^d(s_2)) = C^d(V^d(s_2)) = s_2$, (ii) otherwise, it adds to $P_{K'}^{\mathtt{fin}}$ the tuple $(o,\ pr,\ n(s,d,d'),\ order_1\ N(s_1,d,d'),\ order_2\ n(s_2,d,d'))$, due to rules $R1$ and $R2$ of Definition 9.

Further, for each tuple $(o,\ pr,\ s,\ (s_1, u)) \in P_K^{\mathtt{fin}}$, it adds to $P_{K'}^{\mathtt{fin}}$ the tuple $(o,\ pr,\ n(s,d,d'),\ (s_1, u))$. This is due to Rule $R3$ of to Def. 9 and it means that each datatype instance tuple $(o,\ pr,\ n(s,d,d'),\ s')$, where $s' \in (C^{d'} \setminus n(L^{D(u)}, D(u), d')) \cup \{n(s_1, D(u), d')\}$, belongs to $P_{K'}$.

Then, for all $pr \in Pr_{K'}^{XSD} \setminus Pr_K^{XSD}$ and $(o\ pr\ s\hat{}\hat{}ran_{K'}(pr)) \in K'$, it adds to $P_{K'}^{\mathtt{fin}}$ the tuple $InitDatatype(K', (o\ pr\ ran_{K'}(pr))$. Finally, the set $P_{K'}^{\mathtt{fin}}$ is returned.

[19] Note that in the algorithm, we use the functions $n(s,d,d')$ and $N(s,d,d')$, which have been defined at the end of Section 4.

Example 10. Consider KBs K, K' s.t. $K \rightsquigarrow K'$ and a property $hasWeight \in Pr_K^{XSD}$ s.t. $ran_K(hasWeight) = ex{:}integerTotalDigits3$, which is derived from $xsd{:}integer$ by setting the facet $totalDigits = 3$. Additionally, $ran_{K'}(hasWeight) = ex{:}decimalTotalDigits5$, which is derived from $xsd{:}decimal$ by setting the facet $totalDigits = 5$. Assume that $(o, hasWeight, \text{“9”}, \text{“} > 8\text{”}, \text{“} < 10\text{”}) \in P_K^{\mathtt{fin}}$. Then, $(o, hasWeight, \text{“9”}, \text{“} > 8\text{”}, \text{“} < 10\text{”}) \in P_{K'}^{\mathtt{fin}}$. Note that $PossRep_{K'}((o, hasWeight, \text{“9”}, \text{“} > 8\text{”}, \text{“} < 10\text{”})) = \{(o, hasWeight, \text{“9”}, C^d(v)) \mid v \in V^d \text{ and } 8 < v < 10\}$, where $d = decimalTotalDigits5$. Additionally, assume that $(o, hasWeight, \text{“999”}, \text{“} > 998\text{”}, \text{“} \leq \text{HIGH”}) \in P_K^{\mathtt{fin}}$. Then, $t = (o, hasWeight, \text{“999”}, \text{“} > 998\text{”}, \text{“} \leq \text{HIGH”}) \in P_{K'}^{\mathtt{fin}}$. Note that $PossRep_{K'}(t) = \{(o, hasWeight, \text{“999”}, C^d(v) \mid v \in V^d \text{ and } 998 < v \leq 99999\}$. □

Example 11. Consider KBs K, K' s.t. $K \rightsquigarrow K'$ and a property $hasColour \in Pr_K^{XSD}$ s.t. $ran_K(hasColour) = ex{:}six\_colours$, which is derived from $xsd{:}string$ through the enumeration of the six values “blue”, “green”, “red”, “yellow”, “black”, and “white”. Additionally, $ran_{K'}(hasColour) = ex{:}eight\_colours$, which is derived from $xsd{:}string$ through the enumeration of the six values “blue”, “green”, “red”, “yellow”, “black”, “white”, “magenta”, and “grey”. Assume that $(o, hasColour, \text{“}red\text{”}, (\text{“}red\text{”}, ex{:}six\_colours)) \in P_K^{\mathtt{fin}}$. Then, $t = (o, hasColour, \text{“}red\text{”}, (\text{“}red\text{”}, ex{:}six\_colours)) \in P_{K'}^{\mathtt{fin}}$. Note that $PossRep_{K'}(t) = \{(o, hasColour, \text{“}red\text{”}, v) \mid v \in \{\text{“}red\text{”}, \text{“}magenta\text{”}, \text{“}grey\text{”}\}\}$. □

Example 12. Consider KBs K, K' s.t. $K \rightsquigarrow K'$ and a property $hasName \in Pr_K^{XSD}$ s.t. $ran_K(hasName) = ex{:}stringMax20$, which is derived from $xsd{:}string$ by setting the facet $maxLength = 20$. Additionally, $ran_{K'}(hasName) = ex{:}stringMax30$, which is derived from $xsd{:}string$ by setting the facet $maxLength = 30$. Assume that $(o, hasName, \text{“}Dom.\,Theotocopoulos\text{”}, (\text{“}Dom.\ Theotocopoulos\text{”}, ex{:}stringMax20)) \in P_K^{\mathtt{fin}}$. Then, $t = (o, hasName, \text{“}Dom.\,Theotocopoulos\text{”}, (\text{“}Dom.\ Theotocopoulos\text{”}, ex{:}stringMax20)) \in P_{K'}^{\mathtt{fin}}$. Note that $PossRep_{K'}(t) = \{(o, hasName, \text{“}Dom.\ Theotocopoulos\text{”}, v) \mid v \in C^{stringMax30} \setminus C^{stringMax20} \cup \{\text{“}Dom.\ Theotocopoulos\text{”}\}\}$. □

Let K, K' be KBs s.t. $K \rightsquigarrow K'$. The following proposition shows that Algorithm $DatatypePossibilities(K, K', P_K^{\mathtt{fin}})$ computes the correct set $P_{K'}^{\mathtt{fin}}$, i.e. the one derived from Def. 9 and the fact that $P_{K'} = DIT_{K'} \setminus F_{K'}$.

Prop. 9 Let K, K' be KBs s.t. $K \rightsquigarrow K'$. Then, $P_{K'}^{\mathtt{fin}} = DatatypePossibilities(K, K', P_K^{\mathtt{fin}})$. □

The following proposition provides the time complexity of Algorithm 4, $DatatypePossibilities(K, K', P_K^{\mathtt{fin}})$.

Prop. 10 The time complexity of Algorithm $DatatypePossibilities(K, K', P_K^{\mathtt{fin}})$ is in $O(\,|I_{K'}^{XSD}| * L)$, where L is the length of the longest datatype derivation chain for the XSD datatypes, considered in K'. □

7 Accuracy Life Cycle Management Process

So far, we have seen algorithms for computing the set of compact possible datatype tuples. Now, we will focus on the exploitation of these tuples such that the *accuracy*

of the XSD-typed literal values of property instance triples, affected during ontology evolution, is increased. Let K, K' be KBs s.t. $K \rightsquigarrow K'$ and that our machinery has computed the set $P^{\mathtt{fin}}_{K'}$, using Algorithm 4. Obviously, in this case K' does not satisfy the *MLSA*. Therefore, we need a process that helps the curator of K' to modify K' s.t. K' satisfies the *MLSA* for the XSD-valued instance triples that the curator wishes. To achieve this the following iterative process starts:

1. The curator selects a property $pr \in Pr^{XSD}_{K'}$. Let $d = D(ran_{K'}(pr))$.
2. The system displays the RDF triples $t = (o\ pr\ s\hat{}\hat{}ran_{K'}(pr)) \in I^{XSD}_{K'}$ that do not satisfy the *MLSA*, that is $InitDatatype(K', t) \notin P^{\mathtt{fin}}_{K'}$ [20].
3. The curator selects one of the displayed RDF triples, e.g. $(o\ pr\ s\hat{}\hat{}ran_{K'}(pr))$.
4. The system displays the tuple $t = (o, pr, L2C^d(s), s_1, s_2) \in P^{\mathtt{fin}}_{K'}$.
5. The curator selects a lexical value s' s.t. $(o, pr, L2C^d(s), L2C^d(s')) \in PossRep_{K'}(t)$.
6. The system updates K' by substituting the RDF triple $(o\ pr\ s\hat{}\hat{}ran_{K'}(pr)) \in K'$ by the RDF triple $(o\ pr\ s'\hat{}\hat{}ran_{K'}(pr))$.
7. The system updates $P^{\mathtt{fin}}_{K'}$ by substituting $t \in P^{\mathtt{fin}}_{K'}$ by $InitDatatype(K', (o\ pr\ s'\hat{}\hat{}ran_{K'}(pr)))$.
8. The curator can go back to Step 1 or Step 3, or he/she can finalize the curation process.

Now, when the new version K'' of K' is generated s.t. $K' \rightsquigarrow K''$, Algorithm $DatatypePossibilities(K', K'', P^{\mathtt{fin}}_{K'})$ can be called for generating the new $P^{\mathtt{fin}}_{K''}$. Then, a new curation process may start, again.

Example 13. Consider Example 10. Then, after running Algorithm 4, the following curation process may start:

1. The curator selects the property $hasWeight \in Pr^{XSD}_{K'}$.
2. The system displays the RDF triples in $I^{XSD}_{K'}$ referring to property *hasWeight* that do not satisfy the *MLSA*, including $(o\ hasWeight$ "9"$\hat{}\hat{}ex{:}decimalTotalDigits5)$.
3. The curator selects the RDF triple $(o\ hasWeight$ "9"$\hat{}\hat{}ex{:}decimalTotalDigits5)$.
4. The system displays the tuple $t = (o, hasWeight,$ "9", " > 8", " < 10") $\in P^{\mathtt{fin}}_{K'}$.
5. The curator selects a lexical value "$+8.8$". Note that $(o, hasWeight,$ "9", "8.8") $\in PossRep_{K'}(t)$.
6. The system updates K' by removing the RDF triple $(o\ hasWeight$ "9"$\hat{}\hat{}ex{:}decimalTotalDigits5)$ from K' and adding the RDF triple $(o\ hasWeight$ " $+ 8.8$"$\hat{}\hat{}ex{:}decimalTotalDigits5)$.
7. The system updates $P^{\mathtt{fin}}_{K'}$ by removing t from $P^{\mathtt{fin}}_{K'}$ and adding $InitDatatype(K', (o\ hasWeight$ "$+8.8$"$\hat{}\hat{}decimalTotalDigits5)) = (o, hasWeight,$ "8.8", " > 8.7999", " < 8.8001"). □

Example 14. Consider Example 11. Then, after running Algorithm 4, the following curation process may start:

1. The curator selects the property $hasColour \in Pr^{XSD}_{K'}$.
2. The system displays the RDF triples in $I^{XSD}_{K'}$ referring to property *hasColour* that do not satisfy the *MLSA*, including $(o\ hasColour$ "red"$\hat{}\hat{}ex{:}eight\_colours)$.
3. The curator selects the RDF triple $(o\ hasColour$ "red"$\hat{}\hat{}ex{:}eight\_colours)$.
4. The system displays the tuple $t = (o, hasColour,$ "red",("red",$ex{:}six\_colours)) \in P^{\mathtt{fin}}_{K'}$.

[20] This means that if $t' \in P^{\mathtt{fin}}_{K'}$ s.t. t' corresponds to t then $PossRep_{K'}(t') \neq \{(o, pr, L2C^d(s), L2C^d(s))\}$.

5. The curator selects a lexical value "*magenta*". Note that $(o, hasColour, \text{“}red\text{”}, \text{“}magenta\text{”}) \in PossRep_{K'}(t)$.
6. The system updates K' by removing the RDF triple (o *hasColour* “*red*”^^*ex*:*eight_colours*) from K' and adding the RDF triple (o *hasColour* “*magenta*”^^*ex*:*eight_colours*).
7. The system updates $P^{\mathtt{fin}}_{K'}$ by removing t from $P^{\mathtt{fin}}_{K'}$ and adding $InitDatatype(K', (o\ hasColour\ \text{“}magenta\text{”}\text{^^}ex{:}eight\_colours)) = (o, hasWeight, \text{“}magenta\text{”}, (\text{“}magenta\text{”}, ex{:}eight\_colours)$. □

Example 15. Consider now Example 12. Then, after running Algorithm 4, the following curation process may start:

1. The curator selects the property $hasName \in Pr^{XSD}_{K'}$.
2. The system displays the RDF triples in $I^{XSD}_{K'}$ referring to property *hasName* that do not satisfy the *MLSA*, including (o *hasName* “*Dom. Theotocopoulos*”^^ *ex*:*stringMax30*).
3. The curator selects the RDF triple (o *hasName* “*Dom.Theotocopoulos*”^^ *ex*:*stringMax30*).
4. The system displays the tuple $t = (o, hasName,$ “*Dom.Theotocopoulos*”,(“*Dom. Theotocopoulos*”,*ex*:*stringMax20*)) $\in P^{\mathtt{fin}}_{K'}$.
5. The curator selects a lexical value “*Domenico Theotocopoulos*”. Note that (o, *hasName*, “*Dom. Theotocopoulos*”, “*DomenicoTheotocopoulos*”) $\in PossRep_{K'}(t)$.
6. The system updates K' by removing the RDF triple (o *hasName Dom.* “*Theotocopoulos*” ^^*ex*:*stringMax30*) from K' and adding the RDF triple (o *hasName* “*Domenico Theotocopoulos*” ^^*ex*:*stringMax30*).
7. The system updates $P^{\mathtt{fin}}_{K'}$ by removing t from $P^{\mathtt{fin}}_{K'}$ and adding $InitDatatype(K', (o\ hasName\ \text{“}Domenico\ Theotocopoulos\text{”}\text{^^}ex{:}stringMax30)) = (o, hasName,$ “*Domenico Theotocopoulos*”,(“*Domenico Theotocopoulos*”, *ex*:*stringMax30*)). □

Example 16. Consider now Example 26. Then, after running Algorithm 4, the following curation process may start:

1. The curator selects the property $hasGrade \in Pr^{XSD}_{K'}$.
2. The system displays the RDF triples in $I^{XSD}_{K'}$ referring to property *hasGrade* that do not satisfy the *MLSA*, including (o *hasGrade* “2”^^*ex*:*nine_decimals*).
3. The curator selects the RDF triple (o *hasGrade* “2”^^*ex*:*nine_decimals*).
4. The system displays the tuple $t = (o, hasGrade, \text{“}2\text{”}, \text{“} > 1\text{”}, \text{“} < 3\text{”}) \in P^{\mathtt{fin}}_{K'}$.
5. The curator selects a lexical value 1.5”. Note that $(o, hasGrade, \text{“}2\text{”}, \text{“}1.5\text{”}) \in PossRep_{K'}(t)$.
6. The system updates K' by substituting the RDF triple (o *hasGrade* “2”^^ *ex*:*nine_decimals*) by the RDF triple (o *hasGrade* “1.5”^^ *ex*:*nine_decimals*).
7. The system updates $P^{\mathtt{fin}}_{K'}$ by substituting t by $InitDatatype(K',$ (o *hasGrade* “1.5”^^*ex*:*nine_decimals*) $= (o, hasGrade, \text{“}1.5\text{”}, \text{“} > 1\text{”}, \text{“} < 2\text{”})$. □

The curation process presented so far is a triple-by-triple manually driven fixing activity. However, there may exist an RDF file containing XSD-valued instance triples that satisfy the MLSA, which is automatically parsed (one triple at the time) so as to replace existing data in the KB with more precise information from the RDF file. In particular, assume that the RDF file contains an XSD-valued instance triple (o pr s^^u), where $pr \in Pr^{XSD}_{K'}$ and $D(u)$ is a numeric or dateTime-related XSD datatype. We distinguish two cases (let $d = D(u)$ and $d' = D(ran_{K'}(pr))$:

Case 1: $d' \Rightarrow d$. The system identifies (a) the XSD-valued instance triple $(o\ pr\ s'\hat{}\hat{}\ ran_{K'}(pr)) \in K'$ s.t. $absolute\_distance(n(s', d', d), s) = min(\{absolute\_distance(n(s'', d', d), s) \mid (o\ pr\ s''\hat{}\hat{}ran_{K'}(pr)) \in K'\})$[21] and (b) the closest to $s\hat{}\hat{}u$ literal $s''\hat{}\hat{}ran_{K'}(pr)$, using algorithms similar to these presented in this paper. Then, the system updates K' by substituting the RDF triple $(o\ pr\ s'\hat{}\hat{}ran_{K'}(pr)) \in K'$ by the RDF triple $(o\ pr\ s''\hat{}\hat{}ran_{K'}(pr))$. Finally, the system updates $P^{\mathtt{fin}}_{K'}$ by substituting the tuple $(o, pr, L2C^{d'}(s'), s_1, s_2) \in P^{\mathtt{fin}}_{K'}$ with $InitDatatype(K', (o\ pr\ s''\hat{}\hat{}ran_{K'}(pr)))$.

Case 2: $d \Rightarrow d'$. The system identifies the XSD-valued instance triple $(o\ pr\ s'\hat{}\hat{}ran_{K'}(pr)) \in K'$ s.t. $absolute\_distance(n(s, d, d'), s') = min(\{absolute\_distance(n(s, d, d'), s'') \mid (o\ pr\ s''\hat{}\hat{}ran_{K'}(pr)) \in K'\})$. Now, let $t = (o, pr, L2C^{d'}(s'), s_1, s_2) \in P^{\mathtt{fin}}_{K'}$. If $(o, pr, L2C^{d'}(s'), n(s, d, d')) \in PossRep_{K'}(t)$ then the system updates K' by substituting the RDF triple $(o\ pr\ s'\hat{}\hat{}ran_{K'}(pr)) \in K'$ with the RDF triple $(o\ pr\ n(s, d, d')\hat{}\hat{}ran_{K'}(pr))$[22]. Finally, the system updates $P^{\mathtt{fin}}_{K'}$ by substituting the tuple $(o, pr, L2C^{d'}(s'), s_1, s_2) \in P^{\mathtt{fin}}_{K'}$ by the tuple t, derived as follows: Let $(o, pr, L2C^{d}(s), order_1\ s_1, order_2\ s_2) = InitDatatype(K_{\mathtt{dummy}}, (o\ pr\ s\hat{}\hat{}u))$, where $K_{\mathtt{dummy}}$ is a dummy KB that contains just the triples $(o\ pr\ s\hat{}\hat{}u)$ and $(pr\ rdfs{:}range\ u)$. Then, $t = (o,\ pr,\ n(s, d, d'),\ order_1\ N(s_1, d, d'),\ order_2\ n(s_2, d, d'))$.

Example 17. Consider a KB K that satisfies the *MLSA* and contains the XSD-valued triples $(o\ pr$ "1"$\hat{}\hat{}xsd{:}\ integer)$, $(o\ pr$ "3"$\hat{}\hat{}xsd{:}integer)$, and $(o\ pr$ "5"$\hat{}\hat{}xsd{:}integer)$. Assume that after datatype backwards compatible KB evolution $(K \rightsquigarrow K')$, $ran_{K'}(pr)$ = $ex{:}decimalFractionDigits1$. Now K' contains $(o\ pr$ "1"$\hat{}\hat{}ex{:}decimalFractionDigits1)$, $(o\ pr$ "3"$\hat{}\hat{}ex{:}decimalFractionDigits1)$, and $(o\ pr$ "5"$\hat{}\hat{}ex{:}decimalFractionDigits1)$. Assume now that there exists an RDF file with the information $(o\ pr$ "2.72"$\hat{}\hat{}ex{:}$ *decimalFractionDigits2*). The absolute distance between 1 and 2.72 is 1.72, the absolute distance between 3 and 2.72 is 0.28, and the absolute distance between 5 and 2.72 is 2.28. Thus, the XSD-valued instance triple that will be replaced with a more precise one is $t = (o\ pr$ "3"$\hat{}\hat{}ex{:}decimalFractionDigits1)$. Thus, $t \in K'$ is replaced by $(o\ pr$ "2.7"$\hat{}\hat{}ex{:}decimalFractionDigits1)$ and the tuple $(o, pr,$ "3", " > 2", " < 3") $\in P^{\mathtt{fin}}_{K'}$ is substituted by $(o, pr,$ "2.7", " > 2.6", " < 2.8"). □

Example 18. Consider a KB K' that contains the XSD-valued instance triple $(o\ pr$ "3"$\hat{}\hat{}ex{:}decimalFractionDigits2)$ and assume that $t = (o, pr,$ "3", " > 2", " < 4") $\in P^{\mathtt{fin}}_{K'}$. Assume now that there exists an RDF file with the information $(o\ pr$ "3.2"$\hat{}\hat{}ex{:}$ *decimalFractionDigits1*). Since $(o, pr,$ "3", "3.2") $\in PossRep_{K'}(t)$, the RDF triple $(o\ pr$ "3.2"$\hat{}\hat{}ex{:}decimalFractionDigits2)$ is more precise than the RDF triple $(o\ pr$ "3"$\hat{}\hat{}ex{:}decimalFractionDigits2)$ and it replaces it in K'. Now $t \in P^{\mathtt{fin}}_{K'}$ is substituted by $(o, pr,$ "3.2", " > 3.1", " < 3.3"). □

Note that even if the curator is not able to improve the precision of the XSD-valued instance triples, their computed possibilities provide an important information to the user, as shown in the example at the end of the Introductory Section.

[21] This absolute distance is computed by a simple subtraction on the units of the corresponding XSD datatype.

[22] In this case, the RDF triple $(o\ pr\ n(s, d, d')\hat{}\hat{}ran_{K'}(pr))$ is more "accurate" than the RDF triple $(o\ pr\ s'\hat{}\hat{}ran_{K'}(pr))$.

8 Use Cases

In this Section, we review four use cases of our theory.

Use Case 1: Assume that for some Breast Cancer patients, after undergoing surgery, their tumor is removed and sent to a molecular biology laboratory for taking part in a gene expression profiling experiment, based on the (spotted) DNA microarray technology. In these experiments, fragments of genes, called Reporters, are spotted on a microarray slide, which is hybridized with the cancerous tissue and a "normal" tissue. Hybridization results are analyzed to generate Gene Expression data, expressing through a Ratio Value (per spotted gene), if the gene in the cancerous tissue is over-expressed, under-expressed, or equally expressed with respect to the "normal" tissue. In other words, the gene expression profile of the tumor is compared with that of a "normal" tissue. Consider that the Ratio Value is expressed through the property *ratio_value* and that, in the original KB K, $ran_K(ration\_value) = decimal\_FractionDigits1$. Assume that, in KB K, *ration_value* satisfies the *MLSA*. Now, assume that after datatype backwards compatible KB evolution from KB K to KB K', it holds that $ran_{K'}(ration\_value) = decimal\_FractionDigits2$. Now, in KB K', *ration_value* does not satisfy the *MLSA*. Therefore, the original RDF file containing ration values with three fraction digits should be processed through the automated process described in the previous section such that the ratio values of K' are now updated from having two fraction digits that do not satisfy the *MLSA* to having two fraction digits that satisfy the *MLSA*. Even if the original RDF file does not contain the ratio value for all reporters, the possibilities in $P^{\mathtt{fin}}_{K'}$ contain important information for these cases.

Use Case 2: Consider that the Department of Computer Science and the Archeology Department in Crete, Greece decide to create an interdisciplinary Cultural Informatics Master's program selecting the best thirty student from each department and providing a scholarship to the best five students. However, the grading of the courses in the Department of Computer Science is done based on a scale 0,1,...,10, while grading in the Archeology Department is done based on a scale 0,0.5,...,9.5,10. In order, to make the comparison possible and select the best five student entering the interdisciplinary Master's program, the original RDF files regarding the course grades of the Computer Science students entering the interdisciplinary Master's program should be processed through the automated process described in the previous section such that such that their grades our now updated to the scale 0,0.5,...,9.5,10 and satisfy the *MLSA*. After this updating, comparison of the grades of the students entering the interdisciplinary Master's program is possible and the best five students are selected for taking the scholarship.

Use Case 3: Consider a hospital that uses a system that allows seven letters for the patient's father first name. Assume that the hospital system is upgraded such that it allows fifteen letters for the patient's father first name. Now, the *MLSA* is not satisfied for the first name of the father of some patients. If the *MLSA* is not satisfied for a patient, the next time the patient visits a hospital, it provides its father first name and *MLSA* is restored.

Use Case 4: A company builds a Land Registry (Cadastre) on behalf of the state, which contains legal (e.g. ownership) and technical details (e.g., shape, location, size) of land parcels. This entails the procurement of highly accurate maps which should allow the distinct identification of a property's location, as well as its representation

within the Registry's framework. Orthophotomaps are commonly used because they (i) preserve the geometric features and maintain the qualitative information of photographs, (ii) can be supplemented with additional mapping information such as road networks, real estate properties, local area names and (iii) can easily be used by real estate owners to identify the location of their property, as an orthophotomap represents reality in the same way this is represented in a photograph. Requirements in building the Cadastre are commonly prescribed by a public authority or by law. For example, property boundaries may require a resolution from digital orthophotomaps of $1m^2$/pixel. At a later development phase, however, pragmatic needs may require a more stringent specification, which may lead to more demanding, higher resolution standards, e.g., $1cm^2$/pixel for some selected urban areas.

Through the automated curation process, the updated sizes of the selected urban areas replace previous less accurate sizes, while for the rest of the areas the computed possibilities contain important information.

9 Related Work

Several works exist on schema and ontology evolution (for recent surveys, see [10, 3]). In the following, we briefly describe the most relevant works to the present paper.

Klein et al. [17] propose a versioning mechanism for reducing the problems caused by ontology evolution. They argue that ontology versioning is necessary because changes to ontologies may cause incompatibilities and describe situations where the new (evolved) ontology cannot usefully replace its previous version. They list a number of artifacts that may depend on an ontology, and thus can become incompatible after ontological evolution, such as data conformance to an ontology: when an ontology is changed, data may assume a different interpretation or may use terms that are no longer applicable.

Along the same lines, Xuan et al. [29] propose a model to deal with the problem of asynchronous ontology versions in the context of a materialized integration system, which is based on the principle of *ontological continuity*, which refers to the permanence of classes, properties, and subsumption.

Stojanovic et al. [25] identify a six-phase ontology evolution process and focus on providing the user with control and customization capabilities. In order to enable such customization on the ontology evolution process, the user may choose an advanced evolution strategy. It represents a mechanism to prioritize and arbitrate among different applicable evolution strategies, thus relieving the user from synthesizing an evolution path out of individual strategies. An advanced evolution strategy automatically combines available elementary evolution strategies to satisfy user criteria.

Noy and Klein [22] present an informal discussion on the differences between ontology evolution and database schema evolution, and how structural changes in ontologies affect the preservation of their data instances. They focus on whether instance data can still be accessed through the changed ontology, and they classify the operation effects as information-preserving changes, translatable changes, and information-loss changes.

Konstantinidis et al. [20] focus on the *effects* of a requested change operation, i.e. how the new KB should be after a request for a change, and on its *side-effects* on both schema and instance data, i.e. certain additional actions must be executed to restore validity. They propose a general-purpose algorithm for determining the

effects and side-effects of a requested elementary or complex change operation, so as to resolve the conflicts.

In addition, Qin and Alturi [24] focus on the validity of data instances during ontological evolution. They classify ontology changes into two levels - structural and semantic. Structural changes include changes such as adding/removing classes or properties, modifying the range of a property, etc. Semantic changes mean changes to the semantics presented by the ontology, such as (i) changes to the generalization or descriptiveness of a class or (ii) changes to the generalization or restrictiveness of a property. Semantic changes can be further classified into explicit and implicit changes. While explicit semantic changes can be detected based on the structural changes, implicit semantic changes need to be derived from explicit semantic changes by using implication analysis. They propose an algorithm for evaluating the structural validity of a data instance and then another algorithm for evaluating the semantic validity of a data instance.

Banerjee et al. [1] present a formal framework for schema evolution for the prototype object-oriented database system ORION. In particular, they define the semantics of schema changes using this framework. The framework is based on a graph-theoretic model of the class lattice. It consists of five invariants and twelve rules. Invariants are those properties of a class lattice that must be preserved before and after any schema change. The rules guide the selection of the most meaningful option for preserving the invariants. They also analyze the impact of each schema change on the existing instances.

Curino et al. [4] present a language of schema modification operators to express concisely schema changes on relational databases and tools that allow DB administrators to evaluate the effects of these changes. Additionally, they propose optimized translation of old queries to work on the new schema version and automatic data migration.

Guerrini et al. [9] propose an approach for the incremental validation of XML documents upon schema evolution through a set of evolution primitives. They also propose an efficient adaptation algorithm to make the invalidated document portions conform to the evolved schema.

Klettke [18] shows that XML schema evolution can be realized on a conceptual model. Incremental changes made to the conceptual model result in changes to the associated XML schema and valid documents. Yet, preprocessing on incremental changes is possible. For example, adding a new element and then renaming it, is equivalent to simply adding the element with the new name.

Guerrini and Mesiti [8] present X-Evolution, a Web system developed on top of a commercial DBMS that allows the specification of schema modifications both on a graphical representation of an XML Schema and through a specifically tailored declarative language. X-Evolution supports facilities for performing schema revalidation only when strictly needed and only on the minimal parts of documents affected by the modifications. Moreover, it supports the automatic and query-based adaptation of original schema instances to the evolved schema.

In conclusion, whereas several works and approaches deal with the validity of data during schema and ontology evolution, no work reasons on the *accuracy* of the XSD-typed literal values of property instance triples during datatype evolution. We view our current work as complementary to recent works on fuzzy or probabilistic RDF. Below, we describe some of these approaches.

Udrea et al. [28] introduce a *Probabilistic* RDF framework (for short *pRDF*) for expressing probabilistic information about the relationships expressed in RDF and provide algorithms to efficiently answer queries over *pRDF* ontologies. Mazzieri and Dragoni [21] present an extended syntax to represent fuzzy membership values within RDF statements, and elaborate on their interpretation. Straccia [26] describes a system for a fragment of fuzzy RDF and shows how top-k fuzzy disjunctive queries can be answered by relying on the closure computation and top-k database engines. Huang and Liu [13] present a general framework for supporting SPARQL queries on probabilistic RDF databases and a query evaluation framework based on possible world semantics.

Finally, we should mention the W3C Uncertainty Reasoning for the World Wide Web (URW3) Incubator group[23] whose mission is to better define the challenge of reasoning with and representing uncertain information available in the World Wide Web.

10 Conclusions

Semantic Web ontologies are not static but evolve as the understanding of the domain (or the domain itself) grows or evolves. This evolution is usually independent of the ontological instance descriptions which are stored in Knowledge Bases (KBs). Therefore, the effects of an ontology update on instance descriptions which are migrated to a new ontology version should be studied. In this paper, we are interested in the effects of migrations of literal-valued instance triples of an RDF KB to a new ontology version. As XSD-valued instance triples are migrated from one ontology version to another, their "accuracy" is affected. We manage the quality of these migrated instance triples and design flexible interactive methods through which the curator of the KB can intervene to correct the "inaccuracies" of the literal values that are incurred by the migration process.

Originally, we overview the primitive XSD datatypes and the derived XSD datatypes, the latter being derived from the primitive XSD datatypes or other derived XSD datatypes through facet restriction [23]. We consider RDF KBs that support both primitive and derived XSD datatypes, as presented in [2]. We define datatype evolution (e.g., from integer to decimal, from float to double, from date to dateTime, from a facet-restricted integer to a more general facet-restricted integer, from an enumerated datatype to a more general datatype, etc.) and show how the XSD-valued instance triples are automatically migrated to the new ontology version, where the datatype of their associated property may have evolved.

We define the *Maximum Literal Specificity Assumption* (*MLSA*), which is satisfied for the XSD-valued instance triples that are specified according to the original ontology version or whose property is a new property of the updated ontology. However, the *MLSA* may not be satisfied by the migrated XSD-valued instance triples. We present rules and algorithms that show how the *MLSA* is affected after the XSD-valued instance triple migration. Additionally, we present a curation process through which the curator of the KB can correct the incurred "inaccuracies" by replacing the migrated XSD-valued instance triples with new ones that satisfy the *MLSA*. Our algorithms have linear time and space complexity w.r.t. the size of the XSD-valued

[23] `http://www.w3.org/2005/Incubator/urw3/`

instance triples. Finally, we would like to mention that our work is general and can be applied to any framework considering evolving XSD datatypes.

As future work, we plan to generalize our approach to consider all migrated instance descriptions and not only the XSD-valued ones. Naturally, the specificity problem caused by the migration of the rest of the instance triples is of a different kind and has mainly to do with the fact that (i) a resource that was originally classified to a class may now be more precisely reclassified to a specialized class and (ii) a pair of resources associated by a property may now be more precisely associated by a more specialized property.

Acknowledgements: The authors are grateful to Yannis Tzitzikas for his own ideas and to the anonymous referees for their valuable comments.

References

1. Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *ACM SIGMOD International Conference on Management of Data (SIGMOD-1987)*, pages 311–322, 1987.
2. Jeremy J. Carroll and Jeff Z. Pan. XML Schema Datatypes in RDF and OWL. W3C Working Group Note, 14 March 2006. Available at `http://www.w3.org/TR/swbp-xsch-datatypes/`.
3. Dario Colazzo, Giovanna Guerrini, Marco Mesiti, Barbara Oliboni, and Emmanuel Waller. Document and Schema XML Updates. In *Advanced Applications and Structures in XML Processing: Label Stream, Semantics Utilization and Data Query Technololgies.* IDEA Group, 2010.
4. Carlo Curino, Hyun Jin Moon, and Carlo Zaniolo. Graceful database schema evolution: the PRISM workbench. *34th International Conference on Very Large Data Bases (VLDB-2008)*, 1(1):761–772, 2008.
5. Jos de Bruijn and Stijn Heymans. Logical Foundations of RDF(S) with Datatypes. *Journal of Artificial Intelligence Research (JAIR)*, 38:535–568, 2010.
6. M. Duerst and M. Suignard. RFC 3987: Internationalized Resource Identifiers (IRIs), January 2005. Available at `http://www.ietf.org/rfc/rfc3987.txt`.
7. Tim Bray et al. Extensible Markup Language (XML) 1.1 (Second Edition). W3C Recommendation, 16 August 2006. Available at `http://www.w3.org/TR/xml11/`.
8. Giovanna Guerrini and Marco Mesiti. X-Evolution: A Comprehensive Approach for XML Schema Evolution. In *19th International Workshop on Database and Expert Systems Applications (DEXA-2008)*, pages 251–255, 2008.
9. Giovanna Guerrini, Marco Mesiti, and Matteo Alberto Sorrenti. XML Schema Evolution: Incremental Validation and Efficient Document Adaptation. In *5th International XML Database Symposium on Database and XMLTechnologies, (XSym-2007)*, pages 92–106, 2007.
10. Michael Hartung, James Terwilliger, and Erhard Rahm. Recent Advances in Schema and Ontology Evolution. *Schema Matching and Mapping*, 2011.
11. P. Hayes. RDF Semantics, W3C Recommendation, February 2004. Available at `http://www.w3.org/TR/rdf-mt/`.
12. Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*, pages 1–136. Morgan & Claypool, 2011.
13. Hai Huang and Chengfei Liu. Query Evaluation on Probabilistic RDF Databases. In *10th International Conference on Web Information Systems Engineering (WISE-2009)*, pages 307–320, 2009.

14. IEEE. IEEE Standard for Floating-Point Arithmetic, 29 August 2008. Available at http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4610935&abstractAccess=no&userType=inst.
15. Internet Engineering Task Force (IETF). Best Current Practices 47, 2006. Available at http://tools.ietf.org/rfc/bcp/bcp47.
16. S. Josefsson. RFC 3548: The Base16, Base32, and Base64 Data Encodings, July 2003. Available at http://www.ietf.org/rfc/rfc3548.txt.
17. Michel C. A. Klein and Dieter Fensel. Ontology Versioning on the Semantic Web. In *First Semantic Web Working Symposium (SWWS-2001)*, pages 75–91, 2001.
18. Meike Klettke. Conceptual XML Schema Evolution - the CoDEX Approach for Design and Redesign. In *Workshop Proceedings, Datenbanksysteme in Business, Technologie und Web (BTW 2007)*, pages 53–63, 2007.
19. G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 10 February 2004. Available at http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.
20. G. Konstantinidis, G. Flouris, G. Antoniou, and V. Christophides. A Formal Approach for RDF/S Ontology Evolution. In *18th European Conference on Artificial Intelligence (ECAI-2008)*, pages 70–74, Patras, Greece, July 2008.
21. M. Mazzieri and A. Dragoni. A Fuzzy Semantics for the Resource Description Framework. In *International Semantic Web Conference Workshop in Uncertainty Reasoning for the Semantic Web (URSW 2008)*, pages 244–261. Springer, 2008.
22. Natalya Fridman Noy and Michel C. A. Klein. Ontology Evolution: Not the Same as Schema Evolution. *Knowledge and Information Systems*, 6(4):428–440, 2004.
23. David Peterson, Shudi Gao, Ashok Malhotra, C. M. Sperberg-McQueen, and Henry S. Thompson. W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. W3C Working Draft, 3 December 2009. Available at http://www.w3.org/TR/xmlschema11-2/.
24. Li Qin and Vijayalakshmi Atluri. Evaluating the validity of data instances against ontology evolution over the Semantic Web. *Information & Software Technology*, 51(1):83–97, 2009.
25. Ljiljana Stojanovic, Alexander Maedche, Boris Motik, and Nenad Stojanovic. User-Driven Ontology Evolution Management. In *13th Intern. Conf. on Knowledge Engineering and Knowledge Management (EKAW-2002)*, pages 285–300, 2002.
26. Umberto Straccia. A Minimal Deductive System for General Fuzzy RDF. In *3rd International Conference on Web Reasoning and Rule Systems (RR-2009)*, pages 166–181. Springer, 2009.
27. H. J. ter Horst. Completeness, Decidability and Complexity of Entailment for RDF Schema and a Semantic Extension Involving the OWL Vocabulary. *Journal of Web Semantics*, 3(2-3):79–115, 2005.
28. O. Udrea, VS Subrahmanian, and Z. Majkic. Probabilistic RDF. In *2006 IEEE International Conference on Information Reuse and Integration*, pages 172–177, 2006.
29. D. N. Xuan, L. Bellatreche, and G. Pierra. A Versioning Management Model for Ontology-Based Data Warehouses. In *8th International Conference on Data Warehousing and Knowledge Discovery (DaWaK 2006)*, pages 195–206, 2006.

Appendix A: Primitive XSD Datatypes and Facets

In this Appendix, we present the supported primitive XSD datatypes and their facets.

The supported primitive XSD datatypes are (for more information, see [23]):

- *xsd*:`decimal`: It represents a subset of the real numbers, which can be represented by decimal numerals. The value space of *decimal* is the set of numbers that can be obtained by dividing an integer by a non-negative power of ten, i.e., expressible as $i/10^n$, where i and n are integers and $n \geq 0$. The order relation on *decimal* values is the order relation on real numbers, restricted to this subset. The lexical space of *decimal* is the set of lexical representations which match the regular expression: `(\+|-)?([0-9]+(\.[0-9]*)?|\.[0-9]+)`.
- *xsd*:`float`: The *float* datatype is patterned after the IEEE single-precision 32-bit floating point datatype [14]. Floating point numbers are often used to approximate arbitrary real numbers. The value space of *float* contains the non-zero numbers $m * 2^e$, where m is an integer whose absolute value is less than 2^{24}, and e is an integer between -149 and 104, inclusive. In addition to these values, the value space of *float* also contains the following special values: *positiveZero, negativeZero, positiveInfinity,* and *negativeInfinity*[24]. The lexical representation of the *float* values *positiveZero, negativeZero, positiveInfinity,* and *negativeInfinity* are 0, -0, *INF*, and $-INF$. For the basic values, the order and identity relation on *float* is the order relation for rational numbers. For the special values, it holds that $0 = -0$, *INF* is greater that all *float* values, $-INF$ is less that all *float* values, and 0 and -0 are greater than all the negative numbers and less than all the positive numbers. The lexical space of *float* is the set of lexical representations which match the regular expression:
  `(\+|-)?([0-9]+(\.[0-9]*)?|\.[0-9]+)([Ee](\+|-)?[0-9]+)?`
  `|(\+|-)?INF`
  For example, the string -1.5E2 corresponds to the float that is nearest to the decimal -150. The specific mapping $V^{float}(.)$ is defined in [23].
- *xsd*:`double`: The *double* datatype is patterned after the IEEE double-precision 64-bit floating point datatype [14]. The value space of *double* contains the non-zero numbers $m * 2^e$, where m is an integer whose absolute value is less than 2^{53}, and e is an integer between -1704 and 971, inclusive. In addition to these values, the value space of *double* also contains the following special values: *positiveZero, negativeZero, positiveInfinity,* and *negativeInfinity*[25]. The lexical representation of the special *double* values is as the lexical representation of the corresponding special *float* values.
- *xsd*:`dateTime`: It represents instants of time without a time zone offset[26]. The lexical representation of a *dateTime* value has the form YYYY-MM-DDThh:mm:ss, where *Year* YYYY is an integer, *Month* MM is an integer between 1 and 12, inclusive, *Day* DD is an integer between 1 and 31 inclusive, possibly restricted further depending on month and year, *Hour* hh is an integer between 1 and 23, inclusive, *Minute* mm is an integer between 1 and 59, inclusive, and *Second* ss is an integer between 1 and 59, inclusive[27]. Indeed, the Day value must be no more than 30 if Month is one of 4, 6, 9, or 11; no more than 28 if Month is 2 and year

[24] In this work, we do not consider the special float value *notANumber* (*NaN*).
[25] In this work, we do not consider the special double value *notANumber* (*NaN*).
[26] In this work, we ignore timezone offsets.
[27] In this work, we consider that *Second* is an integer and not a decimal, as in [23].

is not divisible by 4, or is divisible by 100 but not by 400; and no more than 29 if Month is 2 and Year is divisible by 400, or by 4 but not by 100. This is called *Date-of-Month value constraint*. Specifically, the lexical space of *dateTime* is the set of lexical representations which match the regular expression:
`-?([1-9][0-9]{3,}|0[0-9]{3})`
`-(0[1-9]|1[0-2])`
`-(0[1-9]|[12][0-9]|3[01])`
`T(([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9]|(24:00:00))`
as long as the Date-of-Month value constraint holds. For example, the string 2010-02-28T01:20:35 represents the instant of time at 1 hour, 20 minutes, and 35 seconds on 28th of February, 2010. Note that 2010-02-29T01:20:35 is not in the lexical space of *dateTime*, since the Date-of-Month value constraint is not satisfied. Ordering in the value space of *dateTime* is defined in [23].

– *xsd*:`gYear`: It represents Gregorian calendar years. The lexical representation of a *gYear* value has the form YYYY, where *Year* YYYY is an integer. Specifically, the lexical space of *gYear* is the set of lexical representations which match the regular expression: `-?([1-9][0-9]{3,}|0[0-9]{3})`.
– *xsd*:`gYearMonth`: It represents specific whole Gregorian months in specific Gregorian years. The lexical representation of a *gYearMonth* value has the form YYYY-MM, where *Year* YYYY is an integer and *Month* MM is an integer between 1 and 12, inclusive. Specifically, the lexical space of *gYearMonth* is the set of lexical representations which match the regular expression:
`-?([1-9][0-9]{3,}|0[0-9]{3})-(0[1-9]|1[0-2])`.
– *xsd*:`date`: It represents intervals of exactly one day in length on the timeline of *dateTime*, beginning on the beginning moment of each day, up to but not including the beginning moment of the next day). The lexical space of *date* is the same as that for the date part YYYY-MM-DD of *dateTime*.
– *xsd*:`gMonth`: It represents whole (Gregorian) months within an arbitrary year. The lexical representation of a *gMonth* value has the form `--`MM, where *Month* MM is an integer between 1 and 12, inclusive. Specifically, the lexical space of *date* is the set of lexical representations which match the regular expression: `--(0[1-9]|1[0-2])`.
– *xsd*:`gMonthDay`: It represents whole calendar days that recur at the same point in each calendar year, or that occur in some arbitrary calendar year. The lexical representation of a *gMonthDay* value has the form `--`MM-DD, where *Month* MM is an integer between 1 and 12, inclusive and *Day* DD is an integer between 1 and 31, inclusive, s.t. DD must be no more than 30 if MM is one of 4, 6, 9, or 11, and no more than 29 if MM is 2. Specifically, the lexical space of *gMonthDay* is the set of lexical representations which match the regular expression: `--(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])` as long as the previous constraint of *Date* DD is satisfied.
– *xsd*:`gDay`: It represents whole days within an arbitrary month, i.e. days that recur at the same point in each (Gregorian) month. The lexical representation of a *gDay* value has the form `---`DD, where *Day* DD is an integer between 1 and 31, inclusive. Specifically, the lexical space of *gDay* is the set of lexical representations which match the regular expression: `---(0[1-9]|[12][0-9]|3[01])`.
– *xsd*:`time`: It represents instants of time that recur at the same point in each calendar day, or that occur in some arbitrary calendar day. The lexical representation of a *time* value has the form hh:mm:ss, where hh, mm, ss are defined in

the description of *xsd*:`dateTime`. Specifically, the lexical space of *time* is the set of lexical representations which match the regular expression:
`([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9]|(24:00:00)`.

- *xsd*:`boolean`: It represents the values of two-valued logic. The value space of *boolean* is {`true`, `false`}. The lexical space of *boolean* is {`true`, `false`, 1, 0}.
- *xsd*:`hexBinary`: It represents arbitrary hex-encoded binary data. The value space of *hexBinary* is the set of finite-length sequences of zero or more binary octets. The lexical space of *hexBinary* consists of strings of hexadecimal digits, two consecutive digits representing each octet in the corresponding value (treating the octet as the binary representation of a number between 0 and 255). Specifically, the lexical space of *hexBinary* is the set of lexical representations which match the regular expression: `([0-9a-fA-F]{2})*`. For example, 0FB7 is a lexical representation of the two-octet value 00001111 10110111.
- *xsd*:`base64Binary`: It represents arbitrary Base64-encoded binary data. For *base64Binary* data the entire binary stream is encoded using the Base64 encoding defined in [16]. The value space of *base64Binary* is the set of finite-length sequences of zero or more binary octets. The lexical representations of *base64Binary* values are limited to the 65 characters of the Base64 Alphabet defined in [16], i.e., a-z, A-Z, 0-9, the plus sign (+), the forward slash (/) and the equal sign (=), together with the space character (#x20). Specifically, the lexical space of *base64Binary* is the set of lexical representations which match the regular expression:
```
((([A-Za-z0-9+/] ?){4})*
(([A-Za-z0-9+/] ?){3}[A-Za-z0-9+/]|
([A-Za-z0-9+/] ?){2}[AEIMQUYcgkosw048] ?=|
[A-Za-z0-9+/] ?[AQgw] ?= ?=))?
```
Note that each '?' except the last is preceded by a single space character. Additionally, note that this grammar requires the number of non-whitespace characters in the lexical representation to be a multiple of four, and for equal signs to appear only at the end of the lexical representation. To understand the encoding, it is important to note that each character in `[A-Za-z0-9+/]` has a 6-bit binary representation.
- *xsd*:`string`: It represents character strings in XML. The value space of *string* is the set of finite-length sequences of zero or more characters that match the Char production from [7]. Specifically, the lexical space of *string* is the set of lexical representations which match the regular expression:
`([#x1-#xD7FF]|[#xE000-#xFFFD]|[#x10000-#x10FFFF])*`
- *xsd*:`anyURI`: It represents an Internationalized Resource Identifier Reference (IRI). An *anyURI* value can be absolute or relative, and may have an optional fragment identifier (i.e., it may be an IRI Reference). This type should be used when the value fulfills the role of an IRI, as defined in [6] or its successor(s) in the IETF Standards Track. The value space of *anyURI* is the set of finite-length sequences of zero or more characters that match the Char production from [7].

Ordering on the value space of the datatypes *xsd*:`gYear`, *xsd*:`gYearMonth`, *xsd*:`date`, *xsd*:`gMonth`, *xsd*:`gMonthDay`, *xsd*:`gDay`, and *xsd*:`time` is based on the ordering of the value space of the datatype *xsd*:`dateTime` [23]. The value spaces of the datatypes *xsd*:`boolean`, *xsd*:`hexBinary`, *xsd*:`base64Binary`, *xsd*:`string`, and *xsd*:`anyURI` are not ordered.

Below we review the considered constraining facets for deriving a datatype d (see [23]):

– **Length:** It is the number of units of length, where the units of length vary depending on *primAncestor*(d). The value of length must be a non-negative integer. The facet length is associated with the *string, anyURI, hexBinary,* and *base64Binary* datatypes. For the *string* datatype, length is measured in units of characters. For the *anyURI* datatype, length is measured in units of characters (as for string). For the *hexBinary* and the *base64Binary* datatypes, length is measured in octets (8 bits) of binary data. For example, the following is the definition of a datatype derived from *string* that represents product codes which must be exactly 8 characters in length.

```
<simpleType name='productCode'>
   <restriction base='string'>
     <length value='8' fixed='true'/>
   </restriction>
</simpleType>
```

By fixing the value of the length facet (note that `fixed='true'`), we ensure that types derived from `productCode` can change or set the values of other facets, but cannot change the length.

– **minLength:** It is the minimum number of units of length, where the units of length vary depending on *primAncestor*(d). The value of minLength must be a non-negative integer. The facet minLenth is associated with the same datatypes, as the facet length.
– **maxLength:** It is the maximum number of units of length, where the units of length vary depending on *primAncestor*(d). The value of minLength must be a non-negative integer. The facet maxLength is associated with the same datatypes, as the facet length.
– **pattern:** It is a constraint on the value space of a datatype which is achieved by constraining the lexical space to literals which match each member of a set of regular expressions. The facet pattern is associated with the *string, hexBinary, base64Binary,* and *anyURI* datatypes.
– **enumeration:** It constrains the value space to a specified set of values, by enumerating a set of lexical values in the lexical space of the base type. The facet enumeration is associated with all primitive datatypes, except *boolean*.
– **whiteSpace:** It constrains the value space of types derived from *string* such that various behaviors are realized. The value of the whiteSpace facet must be one of *preserve*, *replace*, or *collapse*. If the value of the whiteSpace facet is *preserve* then no normalization is done and the value is not changed. If the value of the whiteSpace facet is *replace* then all occurrences of #x9 (tab), #xA (line feed), and #xD (carriage return) are replaced with #x20 (space). If the value of the whiteSpace facet is *collapse* then after the processing implied by *replace*, contiguous sequences of #x20's are collapsed to a single #x20, and any #x20 at the start or end of the string is removed.
– **maxInclusive:** It is the inclusive upper bound of the value space for an ordered datatype. The value of maxInclusive must be equal to some value in the lexical space of the base type. For example, the following is the definition of a derived datatype which limits values to integers less than or equal to 100.

```
<simpleType name='one-hundred-or-less'>
  <restriction base='integer'>
    <maxInclusive value='100'/>
  </restriction>
```

`</simpleType>`
It is associated with the *decimal, float, double, dateTime, gYear, gYearMonth, date, gMonth, gMonthDay, gDay,* and *time* datatypes.

- **minInclusive:** It is the inclusive lower bound of the value space for an ordered datatype. The value of minInclusive must be equal to some value in the lexical space of the base type. It is associated with the same datatypes as maxInclusive.
- **maxExclusive :** It is the exclusive upper bound of the value space for an ordered datatype. The value of maxExclusive must be equal to some value in the lexical space of the base type. It is associated with the same datatypes as maxInclusive.
- **minExclusive :** It is the exclusive lower bound of the value space for an ordered datatype. The value of minExclusive must be equal to some value in the lexical space of the base type. It is associated with the same datatypes as maxInclusive.
- **totalDigits:** It restricts the magnitude and arithmetic precision of values in the value space of *decimal* and datatypes derived from it. The value of the totalDigits facet must be a positive integer. In particular, if the value of totalDigits is t, the effect is to require that values should be equal to $i/10^n$, for some integers i and n, with $|i| < 10^t$ and $0 \leq n \leq t$. This has as a consequence that the values are expressible using at most t digits in decimal notation.
- **fractionDigits:** It places an upper limit on the arithmetic precision of *decimal* values. The value of fractionDigits must be a non-negative integer. In particular, if the value of fractionDigits is f, then the value space is restricted to values equal to $i/10^n$, for some integers i and n and $0 \leq n \leq f$. This has as a consequence that the values are expressible using at most f digits to the right of the decimal point in decimal notation.

Appendix B: XSD Datatype Facet Constraints

In this Appendix, we present a number of constraints that the values of the XSD datatype facets should satisfy (see [23]).

Constraint 1 (length, minLength, and maxLength) If length is a declared facet of a datatype d with value f then (i) minLength and maxLength cannot be declared facets of d, (ii) there is no ancestor type of d having length is its declared facets, (iii) if minLength is a declared facet of an ancestor type of d with value f' then it should be $f \geq f'$, and (iv) if maxLength is a declared facet of an ancestor type of d with value f' then it should be $f \leq f'$.

Constraint 2 (minLength <= maxLength) If both minLength and maxLength are declared facets of a datatype d then the value of minLength should be less than or equal to the value of maxLength.

Constraint 3 (minLength valid restriction) If minLength is a declared facet of a datatype d with value f then (i) if minLength is a declared facet of an ancestor type of d with value f' then it should be $f \geq f'$ and (ii) if maxLength is a declared facet of an ancestor type of d with value f' then it should be $f \leq f'$.

Constraint 4 (maxLength valid restriction) If maxLength is a declared facet of a datatype d with value f then (i) if maxLength is a declared facet of an ancestor type of d with value f' then it should be $f \leq f'$ and (ii) if minLength is a declared facet of an ancestor type of d with value f' then it should be $f \geq f'$.

Constraint 5 (enumeration valid restriction) If enumeration is a declared facet of a datatype d then (i) if datatype d' is derived from d then enumeration should be a declared facet of d' and $d'.enumeration \subseteq d.enumeration$, and (ii) d should not have any other declared facet.

Constraint 6 (whiteSpace valid restriction) If whiteSpace is a declared facet of a datatype d with values *replace* or *preserve* then whiteSpace cannot be a declared facet of an ancestor type of d with value *collapse*. Additionally, If whiteSpace is a declared facet of a datatype d with value *preserve* then whiteSpace cannot be a declared facet of an ancestor type of d with value *replace*.

Constraint 7 (minInclusive <= maxInclusive) If both minInclusive and maxInclusive are declared facets of a datatype d then the value of minInclusive should be less than or equal to the value of maxInclusive.

Constraint 8 (maxInclusive and minInclusive valid restriction) If maxInclusive or minInclusive is a declared facet of a datatype d with value f then (i) if maxInclusive is a declared facet of an ancestor type of d with value f' then it should hold $f \leq f'$, (ii) if maxExclusive is a declared facet of an ancestor type of d with value f' then it should hold $f < f'$, iii) if minInclusive is a declared facet of an ancestor type of d with value f' then it should hold $f \geq f'$, and (iv) if minExclusive is a declared facet of an ancestor type of d with value f' then it should hold $f > f'$.

Constraint 9 (minExlusive < maxExclusive) If both minExclusive and maxExclusive are declared facets of a datatype d then the value of minExclusive should be less than the value of maxExclusive.

Constraint 10 (maxInclusive and maxExclusive) It is not possible that both maxInclusive and maxExclusive are declared facets of a datatype d.

Constraint 11 (minInclusive and minExclusive) It is not possible that both minInclusive and minExclusive are declared facets of a datatype d.

Constraint 12 (minInclusive < maxExclusive) If both minInclusive and maxExclusive are declared facets of a datatype d then the value of minInclusive should be less than the value of maxExclusive.

Constraint 13 (minExclusive < maxInclusive) If both minExclusive and maxInclusive are declared facets of a datatype d then the value of minExclusive should be less than the value of maxInclusive.

Constraint 14 (maxExclusive valid restriction) If maxExclusive is a declared facet of a datatype d with value f then (i) if maxInclusive is a declared facet of an ancestor type of d with value f' then it should hold $f \leq f'$, (ii) if maxExclusive is a declared facet of an ancestor type of d with value f' then it should hold $f \leq f'$, iii) if minInclusive is a declared facet of an ancestor type of d with value f' then it should hold $f > f'$, and (iv) if minExclusive is a declared facet of an ancestor type of d with value f' then it should hold $f > f'$.

Constraint 15 (minExclusive valid restriction) If minExclusive is a declared facet of a datatype d with value f then (i) if maxInclusive is a declared facet of an

ancestor type of d with value f' then it should hold $f < f'$, (ii) if maxExclusive is a declared facet of an ancestor type of d with value f' then it should hold $f < f'$, iii) if minInclusive is a declared facet of an ancestor type of d with value f' then it should hold $f \geq f'$, and (iv) if minExclusive is a declared facet of an ancestor type of d with value f' then it should hold $f \geq f'$.

Constraint 16 (totalDigits valid restriction) If totalDigits is a declared facet of a datatype d with value f and totalDigits is also a declared facet of an ancestor type of d with value f' then it should hold $f \leq f'$.

Constraint 17 (fractionDigits valid restriction) If fractionDigits is a declared facet of a datatype d with value f and fractionDigits is also a declared facet of an ancestor type of d with value f' then it should hold $f \leq f'$.

Constraint 18 (fractionDigits $\leq$ totalDigits) If fractionDigits is a declared facet of a datatype d with value f and totalDigits is also a declared facet of d or of an ancestor type of d with value f' then it should hold $f \leq f'$.

Appendix C: Derived Built-in XSD Datatypes

In this Appendix, we define the built-in XSD datatypes d that are derived from a primitive XSD datatype d', i.e. $primAncestor(d) = d'$ holds.

First, we present the XSD datatypes d that are derived from decimal, i.e., it holds $primAncestor(d) = decimal$.

- `integer`: It is derived from the *decimal* datatype by setting *fractionDigits* = "0" and disallowing the trailing decimal point.
- `nonPositiveInteger`: It is derived from the *integer* datatype by setting *maxInclusive* = "0".
- `nonNegativeInteger`: It is derived from the *integer* datatype by setting *minInclusive* = "0".
- `PositiveInteger`: It is derived from the *nonNegativeInteger* datatype by setting *minInclusive* = "1".
- `NegativeInteger`: It is derived from the *nonPositiveInteger* datatype by setting *maxInclusive* = " − 1".
- `long`: It is derived from the *integer* datatype by setting *minInclusive* = " − 9223372036854775808" and *maxInclusive* = "9223372036854775807".
- `int`: It is derived from the *long* datatype by setting *minInclusive* = "−2147483648" and *maxInclusive* = "2147483647".
- `short`: It is derived from the *int* datatype by setting *minInclusive* = " − 32768" and *maxInclusive* = "32767".
- `byte`: It is derived from the *short* datatype by setting *minInclusive* = " − 128" and *maxInclusive* = "127".
- `unsignedLong`: It is derived from the *nonNegativeInteger* datatype by setting *maxInclusive* = "18446744073709551615".
- `unsignedInt`: It is derived from the *unsignedLong* datatype by setting *maxInclusive* = "4294967295".
- `unsignedShort`: It is derived from the *unsignedInt* datatype by setting *maxInclusive* = "65535".
- `unsignedByte`: It is derived from the *unsignedShort* datatype by setting *maxInclusive* = "255".

Now, we present the XSD datatypes d that are derived from *string*, i.e., it holds $primAncestor(d) = string$.

- `normalizedString`: It is derived from *string* by setting whitespace="*replace*" and it represents white space normalized strings. The lexical and value space of *normalizedString* is the set of strings that do not contain the carriage return (#xD), line feed (#xA), or tab (#x9) characters.
- `token`: It is derived from *normalizedString* by setting whitespace="*collapse*" and it represents tokenized strings. The lexical and value space of *token* is the set of strings that do not contain the carriage return (#xD), line feed (#xA), or tab (#x9) characters, that have no leading or trailing spaces (#x20), and that have no internal sequences of two or more spaces.
- `language`: It is derived from *token* by setting pattern = `[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*` and it represents formal natural language identifiers, as defined in [15].
- `NMTOKEN`: It is derived from *token* by setting pattern = `\c+` and it represents the NMTOKEN attribute type from [7].
- `Name`: It is derived from *token* by setting pattern = `\i\c*` and it represents the XML names.

Note that `\i` matches any name start character, i.e. a letter, "_", and ":", while `\c` matches any name start character or digit.

Appendix D: Other Algorithms

In this Appendix, we present several algorithms that for readability reasons are not presented in the main paper.

First, we will present Algorithm 5 that for a datatype d s.t. $ordered(d)$ holds, it identifies the values of $d.minInclusive$ or $d.minExclusive$ that are either declared or they are inherited from the ancestor datatypes. Due to the Facet Constraints 8, 11, and 15, we check the hierarchy of datatypes from d to $primAncestor(d)$, and the values of the first declared facets minInclusive or minExclusive that we identify are set as values of $d.minInclusive$ or $d.minExclusive$. Note that due to Facet Constraint 11, it is not possible that both $d.minInclusive$ and $d.minExclusive$ are different than NULL.

Algorithm 5 *IdentifyDatatypeMin*(d)
Input: A datatype d s.t. $ordered(d)$ holds
Output: Identification of the values of the fields $d.minInclusive$ or $d.minExclusive$

```
( 1)  d' := d;
( 2)  While (d' ∉ XSD^prim) do
( 3)     If d'.minInclusive ≠ NULL then
( 4)        d.minInclusive := d'.minInclusive;
( 5)        Return(d);
( 6)     If d'.minExclusive ≠ NULL then
( 7)        d.minExclusive := d'.minExclusive;
( 8)        Return(d);
( 9)     d' := baseType(d);
(10)  endWhile
(11)  Return(d);
```

Now, we present Algorithm 6 that for a datatype d s.t. $ordered(d)$ holds, it identifies the values of $d.maxInclusive$ or $d.maxExclusive$. Due to the Facet Constraints 8, 10, and 14, we check the hierarchy of datatypes from d to $primAncestor(d)$, and the values of the first declared facets maxInclusive or maxExclusive that we identify are set as values of $d.maxInclusive$ or $d.maxExclusive$. Note that due to Facet Constraint 10, it is not possible that both $d.maxInclusive$ and $d.maxExclusive$ are different than NULL.

Algorithm 6 *IdentifyDatatypeMax*(d)
Input: A datatype d s.t. $ordered(d)$ holds
Output: Identification of the values of the fields $d.maxInclusive$ or $d.maxExclusive$

```
( 1)  d' := d;
( 2)  While (d' ∉ XSD^prim) do
( 3)     If d'.maxInclusive ≠ NULL then
( 4)        d.maxInclusive := d'.maxInclusive;
( 5)        Return(d);
( 6)     If d'.maxExclusive ≠ NULL then
( 7)        d.maxExclusive := d'.maxExclusive;
( 8)        Return(d);
( 9)     d' := baseType(d);
(10)  endWhile
(11)  Return(d);
```

Now, we present Algorithm 7 that for a datatype d s.t. $primAncestor(d) = decimal$, it identifies the value of $d.totalDigits$. Due to the Facet Constraint 16, we check the hierarchy of datatypes from d to $decimal$, and the value of the first declared facet totalDigits that we identify is set as value of $d.totalDigits$.

Algorithm 7 *IdentifyTotalDigits*(d)
Input: A datatype d s.t. $primAncestor(d) = decimal$
Output: Identification of value of the field $d.totalDigits$

```
(1)  d' := d;
(2)  While (d' ≠ decimal) do
(3)     If d'.totalDigits ≠ NULL then
(4)        d.totalDigits := d'.totalDigits;
(5)        Return(d);
(6)     d' := baseType(d);
(7)  endWhile
(8)  Return(d);
```

Now, we present Algorithm 8 that for a datatype d s.t. $primAncestor(d) = decimal$, it identifies the value of $d.fractionDigits$. If $d \in derFrom(integer)$ then $d.fractionDigits =$ "0", otherwise due to the Facet Constraint 17, we check the hierarchy of datatypes from d to $decimal$, and the value of the first declared facet fractionDigits that we identify is set as value of $d.fractionDigits$.

Algorithm 8 *IdentifyFractionDigits*(d)
Input: A datatype d s.t. $primAncestor(d) = decimal$
Output: Identification of value of the field $d.fractionDigits$

```
( 1)  If d ∈ derFrom(integer) then
( 2)     d.fractionDigits:="0";
( 3)     Return(d);
( 4)  d' := d;
( 5)  While (d' ≠ decimal) do
( 6)     If d'.fractionDigits ≠ NULL then
( 7)        d.fractionDigits := d'.fractionDigits;
( 8)        Return(d);
( 9)     d' := baseType(d);
(10)  endWhile
(11)  Return(d);
```

Below, we present Algorithm 9, *InitPossDecimal*$(K, (o\ pr\ s\hat{}\hat{}u))$, which takes as input a KB K and an RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$ for which the *MLSA* holds, where $pr \in Pr_K^{XSD}$ s.t. $D(ran_K(pr)).enumeration$ =NULL and $primAncestor(D(ran_K(pr))) = decimal$. The algorithm returns an *MLSA* tuple to be added to $P_K^{\mathtt{fin}}$.

Algorithm 9 *InitPossDecimal*$(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$
Input: A KB K and an RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$ for which the *MLSA* holds, where $pr \in Pr_K^{XSD}$ s.t. $D(ran_K(pr)).enumeration$ =NULL and $primAncestor(D(ran_K(pr))) = decimal$
Output: An *MLSA* tuple to be added to $P_K^{\mathtt{fin}}$

```
( 1)  d := D(ran_K(pr));
( 2)  d := IdentifyDatatypeMin(d);
( 3)  d := IdentifyDatatypeMax(d);
( 4)  d := IdentifyTotalDigits(d);
( 5)  d := IdentifyFractionDigits(d);
( 6)  s' := L2C^d(s);
/* DETERMINING s_1 */
( 7)  If d.minInclusive ≠ NULL and (V^d(s) = V^d(d.minInclusive) or
            ((d.fractionDigits ≠ NULL or d.totalDigits ≠ NULL) and
            previousDecimal(V^d(s), d) < V^d(d.minInclusive))) then
( 8)        s_1 := " ≥LOW";
( 9)  else
(10)     If d.minExclusive ≠ NULL and d.totalDigits ≠ NULL then
(11)        tg := V^integer(d.totalDigits);
(12)        If V^d(d.minExclusive) < -10^tg + 1 then
(13)           If V^d(s) = -10^tg + 1 then
(14)              s_1 := " ≥LOW";
(15)           else
(16)              s_1 := concatString(" > ", C^d(previousDecimal(V^d(s), d)));
(17)        else     /* V^d(d.minExclusive) ≥ -10^tg + 1  */
(18)           If previousDecimal(V^d(s), d) ≤ V^d(d.minExclusive) then
(19)              s_1 := " ≥LOW";
```

(20) else
(21) $s_1 := concatString(\text{“}>\text{”}, C^d(previousDecimal(V^d(s), d)))$;
(22) else /* $d.minExclusive$ = NULL or $d.totalDigits$ = NULL */
(23) If $d.minExclusive \neq$ NULL then
(24) If $d.FractionDigits \neq$ NULL then
(25) If $previousDecimal(V^d(s), d) \leq V^d(d.minExclusive)$ then
(26) $s_1 :=$ “ ≥LOW”;
(27) else
(28) $s_1 := concatString(\text{“}>\text{”}, C^d(previousDecimal(V^d(s), d)))$;
(29) else /* $d.totalDigits$ = NULL and $d.fractionDigits$ = NULL */
(30) $s_1 := concatString(\text{“}\geq\text{”}, s')$;
(31) If $d.totalDigits \neq$ NULL then
(32) $tg := V^{integer}(d.totalDigits)$;
(33) If $V^d(s) := -10^{tg} + 1$ then
(34) $s_1 :=$ “ ≥ LOW”;
(35) else
(36) $s_1 := concatString(\text{“}>\text{”}, C^d(previousDecimal(V^d(s), d)))$;
(37) If $d.minExclusive$ = NULL, $d.totalDigits$ = NULL and
$d.fractionDigits \neq$ NULL then
(38) $s_1 := concatString(\text{“}>\text{”}, C^d(previousDecimal(V^d(s), d)))$;
(39) If $d.minExclusive$ = NULL, $d.totalDigits$ = NULL and
$d.fractionDigits$ = NULL then
(40) $s_1 := concatString(\text{“}\geq\text{”}, s')$;
/* DETERMINING s_2 */
(41) If $d.maxInclusive \neq$ NULL and ($V^d(s) = V^d(d.maxInclusive)$ or
(($d.fractionDigits \neq$ NULL or $d.totalDigits \neq$ NULL) and
$nextDecimal(V^d(s), d) > V^d(d.maxInclusive)$)) then
(42) $s_2 :=$ “ ≤HIGH”;
(43) else
(44) If $d.maxExclusive \neq$ NULL and $d.totalDigits \neq$ NULL then
(45) $tg := V^{integer}(d.totalDigits)$;
(46) If $V^d(d.maxExclusive) > 10^{tg} - 1$ then
(47) If $V^d(s) = 10^{tg} - 1$ then
(48) $s_2 :=$ “ ≤HIGH”;
(49) else
(50) $s_2 := concatString(\text{“}<\text{”}, C^d(nextDecimal(V^d(s), d)))$;
(51) else /* $V^d(d.maxExclusive) \leq 10^{tg} - 1$ */
(52) If $nextDecimal(V^d(s), d) \geq V^d(d.maxExclusive)$ then
(53) $s_2 :=$ “ ≤HIGH”;
(54) else
(55) $s_2 := concatString(\text{“}<\text{”}, C^d(nextDecimal(V^d(s), d)))$;
(56) else /* $d.maxExclusive$ = NULL or $d.totalDigits$ = NULL */
(57) If $d.maxExclusive \neq$ NULL then
(58) If $d.FractionDigits \neq$ NULL then
(59) If $nextDecimal(V^d(s), d) \geq V^d(d.maxExclusive)$ then
(60) $s_2 :=$ “ ≤HIGH”;
(61) else
(62) $s_2 := concatString(\text{“}<\text{”}, C^d(nextDecimal(V^d(s), d)))$;
(63) else /* $d.totalDigits$ = NULL and $d.fractionDigits$ = NULL */
(64) $s_2 := concatString(\text{“}\leq\text{”}, s')$;
(65) If $d.totalDigits \neq$ NULL then
(66) $tg := V^{integer}(d.totalDigits)$;
(67) If $V^d(s) := 10^{tg} - 1$ then

(68) s_2 := " $\leq$ HIGH";
(69) else
(70) $s_2 := concatString(\text{“} < \text{”}, C^d(nextDecimal(V^d(s), d)))$;
(71) If $d.maxExclusive$ = NULL, $d.totalDigits$ = NULL and $d.fractionDigits \neq$ NULL then
(72) $s_2 := concatString(\text{“} < \text{”}, C^d(nextDecimal(V^d(s), d)))$;
(73) If $d.maxExclusive$ = NULL, $d.totalDigits$ = NULL and $d.fractionDigits$ = NULL then
(74) $s_2 := concatString(\text{“} \leq \text{”}, s')$;
(75) Return$((o,\ pr,\ s',\ s_1,\ s_2))$;

Now, we provide the algorithms $previousDecimal(u, d)$ and $nextDecimal(\ u, d)$, used by Algorithm 9, $InitPossDecimal(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$.

First, we present Algorithm 10, $previousDecimal(v, d)$, where d is a datatype s.t. $primAncestor(d) = decimal$ and $d.totalDigits$ or $d.fractionDigits$ are different than NULL. Additionally, v is a decimal value that satisfies the constraints of $d.totalDigits$ and $d.fractionDigits$. The algorithm returns a decimal value v' that is the maximum of the decimal values that are lower than v and satisfy the constraints of $d.totalDigits$ and $d.fractionDigits$, and FAILURE, if no such value exists.

Algorithm 10 $previousDecimal(v, d)$

Input: A datatype d s.t. $primAncestor(d) = decimal$ and $d.totalDigits$ or $d.fractionDigits$ are different than NULL and a decimal value v that satisfies the constraints of $d.totalDigits$ and $d.fractionDigits$

Output: A decimal value v' that is the maximum of the decimal values that are lower than v and satisfy the constraints of $d.totalDigits$ and $d.fractionDigits$, and FAILURE, if no such value exists

(1) If $v > 0$ then
(2) If $d.totalDigits \neq$NULL and $d.fractionDigits \neq$NULL then
(3) If $v > 1$ then
(4) $noDecimalDigits := (log_{10}(v)\ div\ 1) + 1$;
(5) else
(6) $noDecimalDigits := 0$;
(7) $possibleDecimalDigits := V^d(d.totalDigits) - noDecimalDigits$;
(8) If $v > 1$ and $v = 10^{(V^d(d.totalDigits)-1)}$ then
(9) $possibleDecimalDigits := possibleDecimalDigits + 1$;
(10) $decimalDigits = min(\{possibleDecimalDigits, V^d(d.fractionDigits)\})$;
(11) $v' := v - 10^{-decimalDigits}$;
(12) Return(v');
(13) If $d.totalDigits \neq$NULL and $d.fractionDigits =$NULL then
(14) If $v > 1$ then
(15) $noDecimalDigits := (log_{10}(v)\ div\ 1) + 1$;
(16) else
(17) $noDecimalDigits := 0$;
(18) $DecimalDigits := V^d(d.totalDigits) - noDecimalDigits$;
(19) If $v > 1$ and $v = 10^{(V^d(d.totalDigits)-1)}$ then
(20) $decimalDigits := decimalDigits + 1$;
(21) $v' := v - 10^{-decimalDigits}$;
(22) Return(v');

(23) If $d.totalDigits$ =NULL and $d.fractionDigits$ ≠NULL then
(24) $decimalDigits := V^d(fractionDigits)$;
(25) $v' := v - 10^{-decimalDigits}$;
(26) Return(v');
(27) If $v \leq 0$ then
(28) If $d.totalDigits$ ≠NULL and $d.fractionDigits$ ≠NULL then
(29) If $v = -10^{V^d(d.totalDigits)} + 1$ then
(30) Return(FAILURE);
(31) If $v \leq -1$ then
(32) $noDecimalDigits := (log_{10}(-v)\ div\ 1) + 1$;
(33) else
(34) $noDecimalDigits := 0$;
(35) $possibleDecimalDigits := V^d(d.totalDigits) - noDecimalDigits$;
(36) $decimalDigits = min(\{possibleDecimalDigits, V^d(d.fractionDigits)\})$;
(37) $v' := v - 10^{-decimalDigits}$;
(38) Return(v');
(39) If $d.totalDigits$ ≠NULL and $d.fractionDigits$ =NULL then
(40) If $v = -10^{V^d(d.totalDigits)} + 1$ then
(41) Return(FAILURE);
(42) If $v \leq -1$ then
(43) $noDecimalDigits := (log_{10}(-v)\ div\ 1) + 1$;
(44) else
(45) $noDecimalDigits := 0$;
(46) $decimalDigits := V^d(d.totalDigits) - noDecimalDigits$;
(47) $v' := v - 10^{-decimalDigits}$;
(48) Return(v');
(49) If $d.totalDigits$ =NULL and $d.fractionDigits$ ≠NULL then
(50) $decimalDigits := V^d(d.fractionDigits)$;
(51) $v' := v - 10^{-decimalDigits}$;
(52) Return(v');

Algorithm *previousDecimal*(v, d) returns the decimal value v' that is the maximum of the decimal values that are lower than v and satisfies the constraints of $d.totalDigits$ and $d.fractionDigits$. Thus, it returns $v' = v - 10^{-decimalDigits}$, where *decimalDigits* is the maximum number of decimal digits allowed for v'. Note that if $d.totalDigits$ ≠NULL and $v = -10^{V^d(d.totalDigits)} + 1$ then FAILURE is returned, since there is no decimal value lower than v that satisfies the constraint of $d.totalDigits$.

Below, we present Algorithm 11, *nextDecimal*(v, d), where d is a datatype s.t. $primAncestor(d) = decimal$ and $d.totalDigits$ or $d.fractionDigits$ are different than NULL. Additionally, v is a decimal value that satisfies the constraints of $d.totalDigits$ and $d.fractionDigits$. The algorithm returns a decimal value v' that is the minimum of the decimal values that are greater than v and satisfy the constraints of $d.totalDigits$ and $d.fractionDigits$, and FAILURE, if no such value exists.

Algorithm 11 *nextDecimal*(v, d)

Input: A datatype d s.t. $primAncestor(d) = decimal$ and $d.totalDigits$ or $d.fractionDigits$ are different than NULL and a decimal value v that satisfies the constraints of $d.totalDigits$ and $d.fractionDigits$

Output: A decimal value v' that is the minimum of the decimal values that are higher than v and satisfy the constraints of $d.totalDigits$ and $d.fractionDigits$, and FAILURE, if no such value exists

(1) If $v < 0$ then do
(2) If $d.totalDigits \neq$NULL and $d.fractionDigits \neq$NULL then
(3) If $v < -1$ then
(4) $noDecimalDigits := (log_{10}(-v)\ div\ 1) + 1$;
(5) else
(6) $noDecimalDigits := 0$;
(7) $possibleDecimalDigits := V^d(d.totalDigits) - noDecimalDigits$;
(8) If $v < -1$ and $v = -10^{(V^d(d.totalDigits)-1)}$ then
(9) $possibleDecimalDigits := possibleDecimalDigits + 1$;
(10) $decimalDigits = min(\{possibleDecimalDigits, V^d(d.fractionDigits)\})$;
(11) $v' := v + 10^{-decimalDigits}$;
(12) Return(v');
(13) If $d.totalDigits \neq$NULL and $d.fractionDigits =$NULL then
(14) If $v < -1$ then
(15) $noDecimalDigits := (log_{10}(-v)\ div\ 1) + 1$;
(16) else
(17) $noDecimalDigits := 0$;
(18) $DecimalDigits := V^d(d.totalDigits) - noDecimalDigits$;
(19) If $v < -1$ and $v = -10^{(V^d(d.totalDigits)-1)}$ then
(20) $decimalDigits := decimalDigits + 1$;
(21) $v' := v + 10^{-decimalDigits}$;
(22) Return(v');
(23) If $d.totalDigits =$NULL and $d.fractionDigits \neq$NULL then
(24) $decimalDigits := V^d(d.fractionDigits)$
(25) $v' := v + 10^{-decimalDigits}$;
(26) Return(v');
(27) If $v \geq 0$ then
(28) If $d.totalDigits \neq$NULL and $d.fractionDigits \neq$NULL then
(29) If $v = 10^{V^d(d.totalDigits)} - 1$ then
(30) Return(FAILURE);
(31) If $v \geq 1$ then
(32) $noDecimalDigits := (log_{10}(v)\ div\ 1) + 1$;
(33) else
(34) $noDecimalDigits := 0$;
(35) $possibleDecimalDigits := V^d(d.totalDigits) - noDecimalDigits$;
(36) $decimalDigits = min(\{possibleDecimalDigits, V^d(d.fractionDigits)\})$;
(37) $v' := v + 10^{-decimalDigits}$;
(38) Return(v');
(39) If $d.totalDigits \neq$NULL and $d.fractionDigits =$NULL then
(40) If $v = 10^{V^d(d.totalDigits)} - 1$ then
(41) Return(FAILURE);
(42) If $v \leq 1$ then
(43) $noDecimalDigits := (log_{10}(v)\ div\ 1) + 1$;

(44) else
(45) $noDecimalDigits := 0$;
(46) $decimalDigits := V^d(d.totalDigits) - noDecimalDigits$;
(47) $v' := v + 10^{-decimalDigits}$;
(48) Return(v');
(49) If $d.totalDigits$ =NULL and $d.fractionDigits \neq$NULL then
(50) $decimalDigits := V^d(d.fractionDigits)$
(51) $v' := v + 10^{-decimalDigits}$;
(52) Return(v');

Algorithm $nextDecimal(v, d)$ returns the decimal value v' that is the minimum of the decimal values that are greater than v and satisfies the constraints of $d.totalDigits$ and $d.fractionDigits$. Thus, it returns $v' = v+10^{-decimalDigits}$, where $decimalDigits$ is the maximum number of decimal digits allowed for v'. Note that if $d.totalDigits \neq$ NULL and $v = 10^{V^d(d.totalDigits)} - 1$ then FAILURE is returned, since there is no decimal value greater than v that satisfies the constraint of $d.totalDigits$.

Below, we present Algorithm 12, $InitPossFloat(K, pr, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$, which takes as input a KB K and an RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$ for which the $MLSA$ holds, where $pr \in Pr_K^{XSD}$ s.t. $D(ran_K(pr)).enumeration$ =NULL and $primAncestor(D(ran_K(pr))) = float$. The algorithm returns an $MLSA$ tuple to be added to $P_K^{\mathtt{fin}}$.

Algorithm 12 $InitPossFloat(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$
Input: A KB K and an RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$ for which the $MLSA$ holds, where $pr \in Pr_K^{XSD}$ s.t. $D(ran_K(pr)).enumeration$ =NULL and $primAncestor(D(ran_K(pr))) = float$
Output: An $MLSA$ tuple to be added to $P_K^{\mathtt{fin}}$

(1) $d := D(ran_K(pr))$;
(2) $d := IdentifyDatatypeMin(d)$;
(3) $d := IdentifyDatatypeMax(d)$;
(4) $s' := L2C^d(s)$;
/* DETERMINING s_1 */
(5) If $d.minInclusive \neq$ NULL then
(6) If $V^d(s) = V^d(d.minInclusive)$ then
(7) $s_1 :=$ " $\geq$LOW";
(8) else
(9) $s_1 := concatString(" > ", C^d(previousFloat(V^d(s)))$;
(10) else
(11) If $d.minExclusive \neq$ NULL then
(12) If $V^d(d.minExclusive) = previousFloat(V^d(s))$ then
(13) $s_1 :=$ " $\geq$LOW";
(14) else
(15) $s_1 := concatString(" > ", C^d(previousFloat(V^d(s)))$;
(16) else /* $d.minInclusive$ = NULL and $d.minExclusive$ = NULL */
(17) If $V^d(s) = negativeInfinity$ then
(18) $s_1 :=$ " $\geq$LOW";
(19) else
(20) $s_1 := concatString(" > ", C^d(previousFloat(V^d(s)))$;
/* DETERMINING s_2 */

(21) If $d.maxInclusive \neq$ NULL then
(22) If $V^d(s) = V^d(d.maxInclusive)$ then
(23) $s_2 :=$ " $\leq$HIGH";
(24) else
(25) $s_2 := concatString(" < ", C^d(nextFloat(V^d(s)))$;
(26) else
(27) If $d.maxExclusive \neq$ NULL then
(28) If $V^d(d.maxExclusive) = nextFloat(V^d(s))$ then
(29) $s_2 :=$ " $\leq$HIGH";
(30) else
(31) $s_2 := concatString(" < ", C^d(nextFloat(V^d(s)))$;
(32) else /* $d.maxInclusive =$ NULL and $d.maxExclusive =$ NULL */
(33) If $V^d(s) = positiveInfinity$ then
(34) $s_2 :=$ " $\leq$HIGH";
(35) else
(36) $s_2 := concatString(" < ", C^d(nextFloat(V^d(s)))$;
(37) Return$((o,\ pr,\ s',\ s_1,\ s_2))$;

Initially, Algorithm $InitPossFloat(K, (o\ pr\ s\hat{}\hat{}u))$ sets $d = D(ran_K(pr))$ and calls the algorithms $IdentifyDatatypeMin(d)$ and $IdentifyDatatypeMax(d)$, assigning the returned value to d. Then, it sets $s' = L2C^d(s)$. If (i) $d.minInclusive \neq$NULL and $V^d(s) = V^d(d.minInclusive)$, (ii) $d.minExclusive \neq$NULL and $V^d(d.minExclusive) = previousFloat(V^d(s))$[28], or (iii) $d.minInclusive =$NULL, $d.minExclusive =$NULL, and $V^d(s) = positiveInfinity$ then s_1 is set equal to " $\geq$LOW". In all other cases, s_1 is set equal to $concatString(" > ", C^d(previousFloat(V^d(s)))$. Similar is the case for defining s_2. Finally, the tuple $(o,\ pr,\ s',\ s_1,\ s_2)$ is returned to be added to $P_K^{\mathtt{fin}}$.

Example 19. Consider a KB K and a property $hasValue \in Pr_K^{XSD}$ s.t. $ran_K(hasValue) = xsd{:}float$. Note that $L2C^{float}(\text{"1"}) = \text{"1.0}E0\text{"}$. Then, $InitPossFloat(\ K, (o\ \ hasValue\ \text{"1"}\hat{}\hat{}xsd{:}float))$=$(o, hasValue,\ \text{"1.0}E0\text{"}, \text{" } > \ 0.99999994E0\text{"}, \text{" } < \ 1.000001E0\text{"})$[29]. Note that $PossRep_K((o, hasValue, \text{"1.0}E0\text{"}, \text{" } > 0.99999994E0\text{"}, \text{" } < 1.000001E0\text{"})) = \{(o, hasValue, \text{"1.0}E0\text{"}, \text{"1.0}E0\text{"})\}$. □

Below, we present Algorithm 13, $InitPossDate(K, (o\ \ pr\ \ s\hat{}\hat{}ran_K(pr)))$, which takes as input a KB K and an RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$ for which the *MLSA* holds, where $pr \in Pr_K^{XSD}$ s.t. $D(ran_K(pr)).enumeration =$NULL and $primAncestor(D(ran_K(pr))) = date$. The algorithm returns an *MLSA* tuple to be added to $P_K^{\mathtt{fin}}$.

Algorithm 13 $InitPossDate(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$
Input: A KB K and an RDF triple $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$ for which the *MLSA* holds, where $pr \in Pr_K^{XSD}$ s.t. $D(ran_K(pr)).enumeration =$NULL and $primAncestor(D(ran_K(pr))) = date$
Output: An *MLSA* tuple to be added to $P_K^{\mathtt{fin}}$

[28] Note that $previousFloat(v)$ returns the greatest value of the float values that are lower than v and $nextFloat(v)$ returns the smallest value of the float values that are higher than v.
[29] Note that $previousFloat(1) = 0.99999994$ and $nextFloat(1) = 1.000001$.

(1) $d := D(ran_K(pr))$;
(2) $d := IdentifyDatatypeMin(d)$;
(3) $d := IdentifyDatatypeMax(d)$;
(4) $s' := L2C^d(s)$;
/* DETERMINING s_1 */
(5) If $d.minInclusive \neq$ NULL and $V^d(s) = V^d(d.minInclusive)$ then
(6) $s_1 :=$ " $\geq$LOW";
(7) else
(8) If $d.minExclusive \neq$ NULL then
(9) If $V^d(d.minExclusive) = previousDate(V^d(s))$ then
(10) $s_1 :=$ " $\geq$LOW";
(11) else
(12) $s_1 := concatString(" > ", C^d(previousDate(V^d(s)))$;
(13) else /* $d.minExclusive =$ NULL and ($d.minInclusive =$ NULL
(14) or $V^d(s) \neq V^d(d.minInclusive)$) */
(15) $s_1 := concatString(" > ", C^d(previousDate(V^d(s)))$;
/* DETERMINING s_2 */
(16) If $d.maxInclusive \neq$ NULL and $V^d(s) = V^d(d.maxInclusive)$ then
(17) $s_2 :=$ " $\leq$HIGH";
(18) else
(19) If $d.maxExclusive \neq$ NULL then
(40) If $V^d(d.maxExclusive) = nextDate(V^d(s))$ then
(41) $s_2 :=$ " $\leq$HIGH";
(42) else
(43) $s_2 := concatString(" < ", C^d(nextDate(V^d(s)))$;
(44) else /* $d.maxExclusive =$ NULL and ($d.maxInclusive =$ NULL
(45) or $V^d(s) \neq V^d(d.maxInclusive)$) */
(46) $s_2 := concatString(" < ", C^d(nextDate(V^d(s)))$;
(47) Return($(o,\ pr,\ s',\ s_1,\ s_2)$);

Initially, Algorithm $InitPossDate(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$ sets $d = D(ran_K(pr))$ and calls the algorithms $IdentifyDatatypeMin(d)$ and $IdentifyDatatypeMax(d)$, assigning the returned value to d. Then, it sets $s' = L2C^d(s)$. If (i) $d.minInclusive$ $\neq$NULL and $V^d(s) = V^d(d.minInclusive)$ or (ii) $d.minExclusive \neq$NULL and $V^d(d.minExclusive) = previousDate(V^d(s))$[30] then s_1 is set equal to " $\geq$LOW". In all other cases, s_1 is set equal to $concatString(" > ", C^d(previousDate(V^d(s)))$. Similar is the case for defining s_2. Finally, the tuple $(o,\ pr,\ s',\ s_1,\ s_2)$ is returned to be added to the initial $P_K^{\mathtt{fin}}$.

Example 20. Consider a KB K and a property $hasDate \in Pr_K^{XSD}$ s.t. $ran_K(hasDate)$ $= xsd{:}date$. Note that $L2C^{date}(\text{"}1999-03-15\text{"}) = \text{"}1999-03-15\text{"}$.
Then, $InitPossDate(K, (o\ hasDate\ \text{"}1999{-}03{-}15\text{"}\hat{}\hat{}xsd{:}date))$=$(o, hasDate, \text{"}1999-03-15\text{"}, \text{" } > 1999-03-14\text{"}, \text{" } < 1999-03-16\text{"})$. Note that $PossRep_K((o, hasDate, \text{"}1999-03-15\text{"}, \text{" } > 1999-03-14\text{"}, \text{" } < 1999-03-16\text{"})) = \{(o, hasDate, \text{"}1999-03-15\text{"}, \text{"}1999-03-15\text{"})\}$. □

[30] Note that $previousDate(v)$ returns the greatest value of the *date* values that are lower than v and $nextDate(v)$ returns the smallest value of the *date* values that are higher than v.

Appendix E: Proofs

In this Appendix, we provide the proof of the Lemmas and Propositions that appear in the main paper.

Lemma 1 Let K be a KB and $pr \in Pr_K^{XSD}$. If $(o\ pr\ s\hat{}\hat{}ran_K(pr)) \in K$ then $(o\ pr\ L2C^d(s)\hat{}\hat{}ran_K(pr)) \in \mathcal{C}(K)$, where $d = D(ran_K(pr))$.
Proof: It follows directly from the fact that $V^d(L2C^d(s)) = V^d(s)$. □

Lemma 2 Let $d_1, d_2 \in XSD$ and let $s \in L^{d_1} \cap L^{d_2}$. Then, the following hold:
1. If $d_1 = d_2 = d$ then $n(s, d, d) = L2C^d(s)$.
2. If $primAncestor(d_1) = primAncestor(d_2) = d$ then $n(s, d_1, d_2) = N(s, d_1, d_2) = L2C^d(s) = L2C^{d_1}(s) = L2C^{d_2}(s)$. □

Proof:
1) Assume that $d_1 = d_2 = d$. Then, $n(s, d, d) = C^d(m(V^d(s), d, d)) = C^d(V^d(s)) = L2C^d(s)$.
2) Assume that $primAncestor(d_1) = primAncestor(d_2) = d$. First, note that $L2C^{d_1}(s) = C^{d_1}(V^{d_1}(s)) = C^d(V^d(s)) = L2C^d(s)$. Similarly, $L2C^{d_2}(s) = L2C^d(s)$. Now, $n(s, d_1, d_2) = C^{d_2}(m(V^{d_1}(s), d_1, d_2)) = C^{d_2}(V^{d_1}(s)) = C^d(V^d(s)) = L2C^d(s)$. Similarly, $N(s, d_1, d_2) = C^{d_2}(M(V^{d_1}(s), d_1, d_2)) = C^{d_2}(V^{d_1}(s)) = C^d(V^d(s)) = L2C^d(s)$. □

Lemma 3 Let $d, d' \in XSD$ s.t. $d \Rightarrow d'$. Then, $ordered(d)$ holds iff $ordered(d')$ holds.
Proof: Since $d \Rightarrow d'$, it holds that either (i) $primAncestor(d) = primAncestor(d')$, or (ii) $primAncestor(d) \rightarrow^* primAncestor(d')$. In both cases, $ordered(d)$ holds iff $ordered(d')$ holds. □

Prop. 1 Let $d_1, d_2, d_3 \in XSD$ s.t. $d_1 \Rightarrow d_2$ and $d_2 \Rightarrow d_3$. Then, it holds that $d_1 \Rightarrow d_3$.
Proof: Since $d_1 \Rightarrow d_2$, it follows that for all $x \in V^{d_1}$, it holds that $m(x, d_1, d_2) \in V^{d_2}$, $M(x, d_1, d_2) \in V^{d_2}$, and (i) $primAncestor(d_1) = primAncestor(d_2)$, or
(ii) $primAncestor(d_1) \rightarrow^* primAncestor(d_2)$.

Since $d_2 \Rightarrow d_3$, it follows that for all $x \in V^{d_2}$, it holds that $m(x, d_2, d_3) \in V^{d_3}$, $M(x, d_2, d_3) \in V^{d_3}$, and (i) $primAncestor(d_2) = primAncestor(d_3)$, or
(ii) $primAncestor(d_2) \rightarrow^* primAncestor(d_3)$.

Let $x \in V^{d_1}$. We will show that $m(x, d_1, d_3) \in V^{d_3}$. Note that $m(x, d_1, d_3) = m(m(x, d_1, d_2), d_2, d_3)$. Since $m(x, d_1, d_2) \in V^{d_2}$, it follows that $m(m(x, d_1, d_2), d_2, d_3) \in V^{d_3}$. Therefore, $m(x, d_1, d_3) \in V^{d_3}$. Similarly, $M(x, d_1, d_3) \in V^{d_3}$.

Assume that $primAncestor(d_1) = primAncestor(d_2)$ and $primAncestor(d_2) = primAncestor(d_3)$ then $primAncestor(d_1) = primAncestor(d_3)$. Assume that $primAncestor(d_1) = primAncestor(d_2)$ and $primAncestor(d_2) \rightarrow^* primAncestor(d_3)$ then $primAncestor(d_1) \rightarrow^* primAncestor(d_3)$. Assume that $primAncestor(d_1) \rightarrow^* primAncestor(d_2)$ and $primAncestor(d_2) = primAncestor(d_3)$ then $primAncestor(d_1) \rightarrow^* primAncestor(d_3)$. Finally, assume that $primAncestor(d_1) \rightarrow^* primAncestor(d_2)$ and $primAncestor(d_2) \rightarrow^* primAncestor(d_3)$ then $primAncestor(d_1) \rightarrow^* primAncestor(d_3)$. Therefore, $d_1 \Rightarrow d_3$. □

Prop. 2 Let K_1, K_2, K_3 be KBs s.t. $K_1 \gg K_2$ and $K_2 \gg K_3$. Then, it holds that $K_1 \gg K_3$.

Proof: Assume that $K_1 \gg K_2$. Then, (i) for all $pr \in Pr_{K_1}^{XSD}$, it holds that $ran_{K_1}(pr) \Rightarrow ran_{K_2}(pr)$ and (ii)

$$I_{K_2}^{XSD} = \{(o\ pr\ n(s, d_1, d_2)\hat{}\hat{}ran_{K_2}(pr)) \mid (o\ pr\ s\hat{}\hat{}ran_{K_1}(pr)) \in I_{K_1}^{XSD}, \\ d_1 = D(ran_{K_1}(pr)),\ \text{and}\ d_2 = D(ran_{K_2}(pr))\} \cup \\ \{(o\ pr\ s\hat{}\hat{}ran_{K_2}(pr)) \in K_2 \mid pr \in Pr_{K_2}^{XSD} \setminus Pr_{K_1}^{XSD}\}$$

Assume that $K_2 \gg K_3$. Then, (i) for all $pr \in Pr_{K_2}^{XSD}$, it holds that $ran_{K_2}(pr) \Rightarrow ran_{K_3}(pr)$ and (ii)

$$I_{K_3}^{XSD} = \{(o\ pr\ n(s, d_2, d_3)\hat{}\hat{}ran_{K_3}(pr)) \mid (o\ pr\ s\hat{}\hat{}ran_{K_2}(pr)) \in I_{K_2}^{XSD}, \\ d_2 = D(ran_{K_2}(pr)),\ \text{and}\ d_3 = D(ran_{K_3}(pr))\} \cup \\ \{(o\ pr\ s\hat{}\hat{}ran_{K_3}(pr)) \in K_3 \mid pr \in Pr_{K_3}^{XSD} \setminus Pr_{K_2}^{XSD}\}$$

Due to Prop. 1, for all $pr \in Pr_{K_1}^{XSD}$, it holds that $ran_{K_1}(pr) \Rightarrow ran_{K_3}(pr)$. Now,

$$I_{K_3}^{XSD} = \{(o\ pr\ n(n(s, d_1, d_2), d_2, d_3)\hat{}\hat{}ran_{K_3}(pr)) \mid (o\ pr\ s\hat{}\hat{}ran_{K_1}(pr)) \in I_{K_1}^{XSD}\} \cup \\ \{(o\ pr\ n(s, d_2, d_3)\hat{}\hat{}ran_{K_3}(pr)) \mid (o\ pr\ s\hat{}\hat{}ran_{K_2}(pr)) \in I_{K_2}^{XSD}, pr \in Pr_{K_2}^{XSD} \setminus Pr_{K_1}^{XSD}\} \cup \\ \{(o\ pr\ s\hat{}\hat{}ran_{K_3}(pr)) \in K_3 \mid pr \in Pr_{K_3}^{XSD} \setminus Pr_{K_2}^{XSD}\}$$

Now, since $(o\ pr\ n(s, d_2, d_3)\hat{}\hat{}ran_{K_3}(pr)) \in K_3$, for $(o\ pr\ s\hat{}\hat{}ran_{K_2}(pr)) \in I_{K_2}^{XSD}$ and $pr \in Pr_{K_2}^{XSD} \setminus Pr_{K_1}^{XSD}$, it follows that

$$I_{K_3}^{XSD} = \{(o\ pr\ n(s, d_1, d_3)\hat{}\hat{}ran_{K_3}(pr)) \mid (o\ pr\ s\hat{}\hat{}ran_{K_1}(pr)) \in I_{K_1}^{XSD}\} \cup \\ \{(o\ pr\ s\hat{}\hat{}ran_{K_3}(pr)) \in K_3 \mid pr \in Pr_{K_3}^{XSD} \setminus Pr_{K_1}^{XSD}\}$$ □

Prop. 3 Let K and K' be two KBs s.t. $K \gg K'$. Let $pr \in Pr_K^{XSD}$ s.t. $d = D(ran_K(pr))$ and $d' = D(ran_{K'}(pr))$.

1. If $d \Rightarrow d'$ and $(o,\ pr,\ s,\ s') \in DIT_K$ then $(o,\ pr,\ n(s, d, d'),\ n(s', d, d')) \in DIT_{K'}$.
2. If $d = d'$ and $(o,\ pr,\ s,\ s') \in DIT_K$ then $(o,\ pr,\ s,\ s') \in DIT_{K'}$.

Proof:
1) Assume that $d \Rightarrow d'$ and $(o,\ pr,\ s,\ s') \in DIT_K$. Then, there is $(o\ pr\ s_1\hat{}\hat{}ran_K(pr)) \in K$, $s = L2C^d(s_1)$, and $s' \in C^d$. Thus, $(o\ pr\ n(s_1, d, d')\hat{}\hat{}ran_{K'}(pr)) \in K'$ and $n(s', d, d') \in C^{d'}$. Since $L2C^{d'}(n(s_1, d, d')) = n(s, d, d)$. It follows that $(o,\ pr,\ n(s, d, d'),\ n(s', d, d')) \in DIT_{K'}$.
2) It follows directly from 1) based on the fact that if $d = d'$ and $s, s' \in C^d$ then $n(s, d, d') = s$ and $n(s', d, d') = s'$. □

Prop. 4 Let $(o, pr, s, s') \in F_K$ s.t. $ordered(d)$ holds, for $d = D(ran_K(pr))$. It holds that:

1. If $V^d(s') < V^d(s)$ then, for all $s'' \in C^d$ s.t. $V^d(s'') \leq V^d(s')$, $(o, pr, s, s'') \in F_K$.
2. If $V^d(s') > V^d(s)$ then, for all $s'' \in C^d$ s.t. $V^d(s'') \geq V^d(s')$, $(o, pr, s, s'') \in F_K$.

Proof:
1) Let $(o, pr, s, s') \in F_K$ s.t. $V^d(s') < V^d(s)$ and $ordered(d)$ holds, for $d = D(ran_K(pr))$. If K satisies the *MLSA* then 1) follows, immediately. Otherwise, it follows from Rule

$R1$ that there is KB K_0 s.t. $K_0 \rightsquigarrow K$, $(o,\ pr,\ s_0,\ s_1) \in F_{K_0}$ and $n(s_0, d_0, d) = s$, where $d_0 = D(ran_{K_0}(pr))$, $V^{d_0}(s_1) < V^{d_0}(s_0)$, and $V^d(s') \leq M(V^{d_0}(s_1), d_0, d)$. Additionally, for all $v \in V^d$ s.t. $v \leq M(V^{d_0}(s_1), d_0, d)$, it holds that $(o,\ pr,\ n(s_0, d_0, d),\ C^d(v)) \in F_K$. Since $V^d(s') \leq M(V^d(s_1), d_0, d)$, and $V^d(s'') \leq V^d(s')$, it follows that $V^d(s'') \leq M(V^{d_0}(s_1), d_0, d)$. Therefore, $(o, pr, s, s'') \in F_{K'}$.

2) It is proved similarly to 1). □

Prop. 5 Let $(o, pr, s, s_1), (o, pr, s, s_3) \in P_K$ s.t. $ordered(d)$ holds, for $d = D(ran_K(pr))$. If $V^d(s_1) < V^d(s_3)$ then, for all $s_2 \in C^d$ s.t. $V^d(s_1) \leq V^d(s_2) \leq V^d(s_3)$, $(o, pr, s, s_2) \in P_K$.

Proof: Assume that $V^d(s_1) \leq V^d(s_2) \leq V^d(s_3)$ and $(o, pr, s, s_2) \in M_K$. If $V^d(s_2) < V^d(s)$ then it follows from Prop. 4 that $(o, pr, s, s_1) \in M_K$, which is impossible. If $V^d(s_2) > V^d(s)$ then it follows from Prop. 4 that $(o, pr, s, s_3) \in M_K$, which is impossible. If $V^d(s_2) = V^d(s)$ then $(o, pr, s, s_2) \in P_K$, which is impossible. □

Prop. 6 Let K, K' be KBs s.t. $K \rightsquigarrow K'$. If $(o, pr, s, s') \in P_K$ then $(o, pr, n(s, d, d'),\ n(s', d, d')) \in P_{K'}$, where $d = D(ran_K(p))$ and $d' = D(ran_{K'}(pr))$.

Proof:

Let $(o, pr, s, s') \in P_K$. Assume that $ordered(d)$ holds and $(o, pr, n(s, d, d'), n(s', d, d')) \in F_{K'}$. Then, there is $(o, pr, s, s'') \in F_K$ s.t. either (i) $V^d(s'') < V^d(s)$ and $m(V^d(s'), d, d') \leq M(V^d(s''), d, d')$ (from Rule $R1$) or (ii) $V^d(s'') > V^d(s)$ and $m(V^d(s'), d, d') \geq m(V^d(s''), d, d')$ (from Rule $R2$). Consider the case where $V^d(s'') < V^d(s)$ and $m(V^d(s'), d, d') \leq M(V^d(s''), d, d')$. It follows from the latter that $V^d(s') \leq V^d(s'')$. Therefore, it follows from Prop. 4.1 that $(o, pr, s, s') \in F_K$, which is impossible. Consider now the case that $V^d(s'') > V^d(s)$ and $m(V^d(s'), d, d') \geq m(V^d(s''), d, d')$. It follows from the latter that $V^d(s') \geq V^d(s'')$. Therefore, it follows from Prop. 4.2 that $(o, pr, s, s') \in F_K$, which is impossible.

Assume now that $ordered(d)$ does not hold and $(o, pr, n(s, d, d'),\ n(s', d, d')) \in F_{K'}$. Then, from Rule $R3$ it follows that $(o, pr, s, s') \in F_K$, which is impossible. □

Prop. 7 Let K_1, K_2, K_3 be KBs s.t. $K_1 \rightsquigarrow K_2 \rightsquigarrow K_3$. Consider the one step transition $(F_{K_1}, P_{K_1}) \rightsquigarrow (F_{K_3}, P_{K_3})$ and the successive step transitions $(F_{K_1}, P_{K_1}) \rightsquigarrow (F'_{K_2}, P'_{K_2}) \rightsquigarrow (F'_{K_3}, P'_{K_3})$. It holds that $P_{K_3} = P'_{K_3}$ and $F_{K_3} = F'_{K_3}$.

Proof:

We consider the following cases:

Case 1) Assume that $pr \in Pr^{XSD}_{K_1}$ s.t. $ordered(D(ran_{K_1}(pr)))$ holds. Let $d_1 = D(ran_{K_1}(pr))$, $d_2 = D(ran_{K_2}(pr))$, and $d_3 = D(ran_{K_3}(pr))$.

Successive step transition: Let $(o, pr, s_1, s'_1) \in F_{K_1}$. Consider first the transition $K_1 \rightsquigarrow K_2$. If Rule $R1$ applies, the following false datatype instance tuples are generated: $(o,\ pr,\ s_2,\ C^{d_2}(v_2)) \in F'_{K_2}$ s.t. $n(s_1, d_1, d_2) = s_2$ and $v_2 \in V^{d_2}$ s.t. $v_2 \leq M(V^{d_1}(s'_1), d_1, d_2)$. Consider now the transition $K_2 \rightsquigarrow K_3$. Then, Rule $R1$ applies again on the previously generated false datatype instance tuples and the following false datatype instance tuples are generated $(o,\ pr,\ s_3,\ C^{d_3}(v_3)) \in F'_{K_3}$ s.t. $n(s_2, d_2, d_3) = s_3$ and $v_3 \in V^{d_3}$ s.t. $v_3 \leq M(v_2, d_2, d_3)$.

Direct step transition: Let $(o, pr, s_1, s'_1) \in F_{K_1}$. Consider the direct step transition $K_1 \rightsquigarrow K_3$. If Rule $R1$ applies then the following false datatype instance tuples are

generated: $(o,\ pr,\ s_3,\ C^{d_3}(v_3)) \in F_{K_2}$ s.t. $n(s_1, d_1, d_3) = s_3$ and $v_3 \in V^{d_3}$ s.t. $v_3 \leq M(V^{d_1}(s'_1), d_1, d_3)$.

Since $K_1 \gg K_2 \gg K_3$, it follows that $d_1 \Rightarrow d_2 \Rightarrow d_3$. Therefore, $n(n(s_1, d_1, d_2), d_2, d_3) = n(s_1, d_1, d_3)$ and $M(M(V^{d_1}(s'_1),\ d_1, d_2), d_2, d_3) = M(V^{d_1}(s'_1), d_1, d_3)$ (see Section 4). From this it follows that the set of false datatype instance tuples that is generated either with the successive step transition or with the direct step transition is the same. The same is true if Rule $R2$ instead of Rule $R1$ applies.

Case 2) Assume that $pr \in Pr^{XSD}_{K_1}$ s.t. $ordered(D(ran_{K_1}(pr))$ does not hold. Let $d_1 = D(ran_{K_1}(pr))$, $d_2 = D(ran_{K_2}(pr))$, and $d_3 = D(ran_{K_3}(pr))$.

Successive step transition: Let $(o, pr, s_1, s'_1) \in F_{K_1}$. Consider the transition $K_1 \rightsquigarrow K_2$. Then, Rule $R3$ applies and the following false datatype instance tuple is generated: $(o,\ pr,\ n(s_1, d_1, d_2), n(s'_1, d_1, d_2)) \in F'_{K_2}$. Consider the transition $K_2 \rightsquigarrow K_3$. Then, Rule $R3$ applies again on the previously generated false datatype instance tuple and the following false datatype instance tuple is generated: $(o,\ pr,\ n(n(s_1, d_1, d_2), d_2, d_3),\ n(n(s'_1, d_1, d_2), d_2, d_3)) \in F'_{K_3}$.

Direct step transition: Let $(o, pr, s_1, s'_1) \in F_{K_1}$. Consider the direct step transition $K_1 \rightsquigarrow K_3$. Then, Rule $R3$ applies and the following false datatype instance tuple is generated: $(o,\ pr,\ n(s_1, d_1, d_3),\ n(s'_1, d_1, d_3)) \in F_{K_2}$.

It follows that that the set of false datatype instance tuples that is generated either with the successive step transition or the direct step transition is the same, based on the observations $n(n(s_1, d_1, d_2), d_2, d_3) = n(s_1, d_1, d_3)$ and $n(n(s'_1, d_1, d_2), d_2, d_3) = n(s'_1, d_1, d_3)$, that hold since $d_1 \Rightarrow d_2 \Rightarrow d_3$.

Case 3) Assume that $pr \in Pr^{XSD}_{K_3} \setminus Pr^{XSD}_{K_1}$ and let $d = D(ran_{K_3}(pr))$.

In this case the false datatype instance tuples, referring to pr, that are generated from the successive step transition are the same as the the false datatype instance tuples, referring to pr, that are generated from the direct step transition and equal to $\{(o, pr, s, s') \in DIT_{K_3} \mid s \neq s'\}$. This is because, each RDF triple $(o\ pr\ s''\hat{}\hat{}ran_{K_3}(pr)) \in I^{XSD}_{K_3}$, where $s = L2C^d(s'')$, satisfies the *MLSA*.

Therefore, it holds that $F_{K_3} = F'_{K_3}$. Additionally, since for a KB K, $P_K = DIT_K \setminus F_K$, it holds that $P_{K_3} = P'_{K_3}$. □

Lemma 4 Let K be a KB. The following hold:

1. For each $t \in I^{XSD}_K$, there is a unique $t' \in P^{\mathtt{fin}}_K$ that corresponds to t.
2. For each $t \in P^{\mathtt{fin}}_K$, there is a unique $t' \in I^{XSD}_K$ s.t. t corresponds to t'.
3. $|P^{\mathtt{fin}}_K| = |I^{XSD}_K|$.

Proof:

1) Let $t = (o\ pr\ s\hat{}\hat{}ran_K(pr)) \in I^{XSD}_K$ and assume that there are two compact possible datatype tuples t_1 and t_2 that correspond to t. Then, $(o, pr, L2C^d(s), L2C^d(s)) \in PossRep(t_1)$ and $(o, pr, L2C^d(s), L2C^d(s)) \in PossRep(t_2)$, for $d = D(ran_K(pr))$. However, this is impossible because it should be $PossRep_K(t_1) \cap PossRep_K(t_2) = \emptyset$.

2) Let $t = (o, pr, s, s_1, s_2) \in P^{\mathtt{fin}}_K$ and assume that there are two RDF triples $t_1 = (o\ pr\ s'_1\hat{}\hat{}ran_K(pr)) \in I^{XSD}_K$ and $t_2 = (o\ pr\ s'_2\hat{}\hat{}ran_K(pr)) \in I^{XSD}_K$ s.t. t corresponds to t_1 and t corresponds to t_2. Then, $L2C^d(s'_1) = L2C^d(s'_2) = s$, for $d = D(ran_K(pr))$.

This means that $V^d(s'_1) = V^d(s'_2)$, which is impossible since K is a valid KB (see Section 4).

3) It follows, directly, from 1) and 2) above. □

Prop. 8 The time complexity of Algorithm *InitDatatypePossibilities*(K) is in $O(|I_K^{XSD}| * L)$, where L is the length of the longest datatype derivation chain for the XSD datatypes, considered in K.

Proof: This is because that time complexity of Algorithm *InitDatatype*$(K, (o\ pr\ s\hat{}\hat{}ran_K(pr)))$ is in $O(L)$ and this algorithm is called by Algorithm *InitDatatypePossibilities*(K) $|I_K^{XSD}|$ times. □

Prop. 9 Let K, K' be KBs s.t. $K \rightsquigarrow K'$. Then, $P_{K'}^{\mathtt{fin}} =$ *DatatypePossibilities*$(K, K', P_K^{\mathtt{fin}})$.

Proof: Let $pr \in Pr_K^{XSD}$, $d = D(ran_K(pr))$, and $d' = D(ran_{K'}(pr))$. Let $(o, pr, s, > s_1, < s_2) \in P_K^{\mathtt{fin}}$. Then, the only tuples of the form (o, pr, s, s') in F_K satisfy that $V^d(s') \leq V^d(s_1)$ or $V^d(s') \geq V^d(s_2)$. Then, due to Def. 9 (Rule $R1$ and Rule $R2$), it follows that the only tuples of the form $(o, pr,\ n(s, d, d'), s'')$ in $F_{K'}$ satisfy that $V^{d'}(s'') \leq M(V^d(s_1), d, d')$ and $V^{d'}(s'') \geq m(V^d(s_1), d, d')$. Thus, $(o, pr, n(s, d, d'), > N(s_1, d, d'), < n(s_2, d, d')) \in P_{K'}^{\mathtt{fin}}$.

Let $(o, pr, s, > s_1, \text{“}\leq\text{HIGH”}) \in P_K^{\mathtt{fin}}$. Then, the only tuples of the form (o, pr, s, s') in F_K satisfy that $V^d(s') \leq V^d(s_1)$. Then, due to Def. 9 (Rule $R1$), it follows that the only tuples of the form $(o, pr,\ n(s, d, d'), s'')$ in $F_{K'}$ satisfy that $V^{d'}(s'') \leq M(V^d(s_1), d, d')$. Thus, $(o, pr, n(s, d, d'), > N(s_1, d, d'), \text{“}\leq\text{HIGH”}) \in P_{K'}^{\mathtt{fin}}$.

Let $(o, pr, s, \geq s_1, \leq s_2) \in P_K^{\mathtt{fin}}$. Then, the only tuples of the form (o, pr, s, s') in F_K satisfy that $V^d(s') < V^d(s_1)$ or $V^d(s') < V^d(s_2)$. It holds that $primAncestor(d) = primAncestor(d') = decimal$, $d.fractionDigits = d'.fractionDigits$=NULL, and $d.totalDigits = d'.totalDigits$=NULL. Therefore, due to Def. 9 (Rule $R1$ and Rule $R2$), it follows that the only tuples of the form $(o, pr,\ n(s, d, d'), s'')$ in $F_{K'}$ satisfy that $V^{d'}(s'') < M(V^d(s_1), d, d')$ and $V^{d'}(s'') < m(V^d(s_1), d, d')$. Thus, $(o, pr, n(s, d, d'), \geq N(s_1, d, d'), \leq n(s_2, d, d')) \in P_{K'}^{\mathtt{fin}}$.

Let $(o, pr, s, \geq s_1, \text{“}\leq\text{HIGH”}) \in P_K^{\mathtt{fin}}$. Then, the only tuples of the form (o, pr, s, s') in F_K satisfy that $V^d(s') < V^d(s_1)$. It holds that $primAncestor(d) = primAncestor(d') = decimal$, $d.fractionDigits = d'.fractionDigits$=NULL, and $d.totalDigits = d'.totalDigits$=NULL. Therefore, due to Def. 9 (Rule $R1$), it follows that the only tuples of the form $(o, pr,\ n(s, d, d'), s'')$ in $F_{K'}$ satisfy that $V^{d'}(s'') < M(V^d(s_1), d, d')$. Thus, $(o, pr, n(s, d, d'), \geq N(s_1, d, d'), \text{“}\leq\text{HIGH”}) \in P_{K'}^{\mathtt{fin}}$.

Similarly, to the previous four cases of tuples in $P_K^{\mathtt{fin}}$, we can prove that if $(o, pr, s, order_1\ s_1, order_2\ s_2) \in P_K^{\mathtt{fin}}$ such that $ordered(d)$ holds then $(o, pr, n(s, d, d'),$ $order_1\ s_1, order_2\ s_2) \in P_{K'}^{\mathtt{fin}}$. Note that, since $d \Rightarrow d'$, if $primAncestor(d) = primAncestor(d')$ then $n(s, d, d') = s$, $N(s_1,\ d, d') = s_1$, and $n(s_2, d, d') = s_2$ (see Lemma 2).

Consider now the final case, where $(o,\ pr,\ s,\ (s_1, u)) \in P_K^{\mathtt{fin}}$. Obviously in this case $ordere(d)$ does not hold. Then, the only tuples of the form (o, pr, s, s') in F_K satisfy that $s' \in n(L^{D(u)}, D(u), d) \backslash \{n(s_1, D(u), d)\}$. Since $u \Rightarrow d \Rightarrow d'$, it follows from Def. 9 (Rule $R3$) that the only tuples of the form $(o, pr,\ n(s, d, d'), s'')$ in $F_{K'}$ satisfy that $s'' \in n(L^{D(u)}, D(u), d') \setminus \{n(s_1, D(u), d')\}$. Thus, $(o,\ pr,\ n(s, d, d'),\ (s_1, u)) \in P_{K'}^{\mathtt{fin}}$.

Let $pr \in Pr^{XSD}_{K'} \setminus Pr^{XSD}_{K}$. Then, for all $t = (o\ pr\ s\hat{}\hat{}ran_{K'}(pr)) \in K'$, Algorithm *InitDatatype*$(K', (o\ pr\ s\hat{}\hat{}ran_{K'}(pr)))$ is called to derive the original compact possible datatype tuple corresponding to t, to be added to $P^{\mathtt{fin}}_{K'}$. Note that since pr is a new property, t satisfies the *MLSA*. □

Proposition 10 The time complexity of Algorithm *DatatypePossibilities*$(K, K', P^{\mathtt{fin}}_{K})$ is in $O(\ |I^{XSD}_{K'}| * L)$, where L is the length of the longest datatype derivation chain for the XSD datatypes, considered in K'.
Proof: The time complexity of lines (1)-(13) of Algorithm *DatatypePossibilities*$(K, K', P^{\mathtt{fin}}_{K})$ is in $O(|P^{\mathtt{fin}}_{K}|) = O(|I^{XSD}_{K}|)$. The time complexity of lines (14)-(16) is in $O(|I^{XSD}_{K'}| * L)$. Since $|I^{XSD}_{K}| < |I^{XSD}_{K'}|$, it follows that the total time complexity of the algorithm is $O(|I^{XSD}_{K'}| * L)$. □

Appendix F: Additional Examples

In this Appendix, we present several examples that for readability reasons are not presented in the main paper.

Example 21. Consider a KB K. Let $pr \in Pr^{XSD}_{K}$ with range *xsd*:*gYearMonth*. Assume that $(o,\ pr, \text{“}2010-01\text{”}, C^{gYearMonth}(v)) \in F_K$, for all $v \in V^{gYearMonth}$ s.t. $v \leq 2009-12$ or $v \geq 2011-01$. Consider now a KB K' s.t. $K \rightsquigarrow K'$. Assume that the range of pr in K' is *xsd*:*date*. Note that $n(\text{“}2010-01\text{”}, gYearMonth, date) =$ “$2010-01-01$”, $M(2009-12, gYearMonth, date) = 2009-12-31$, and $m(2011-01, gYearMonth, date) = 2011-01-01$. Therefore, using rules $R1$ and $R2$, $(o,\ pr, \text{“}2010-01-01\text{”}, C^{date}(v)) \in F_{K'}$, for all $v \in V^{date}$ s.t. $v \leq 2009-12-31$ or $v \geq 2011-01-01$. Additionally, $(o,\ pr, \text{“}2010-01-01\text{”}, C^{date}(v)) \in P_{K'}$, for all $v \in V^{date}$ s.t. $2009-12-31 < v < 2011-01-01$. This means that the RDF triple $(o\ pr\ \text{“}2010-01-01\text{”}\hat{}\hat{}xsd{:}date) \in K'$ can be possibly replaced (after the curation process) by any RDF triple $(o\ pr\ \ s\hat{}\hat{}xsd{:}date)$, where $s \in L^{date}$ s.t. $2009-12-31 < V^{date}(s) < 2011-01-01$. □ □

Example 22. Consider a KB K for which the *MLSA* holds. Assume that the range of property $pr \in Pr^{XSD}_{K}$ is *ex*:*four_decimals*[31], where *ex*:*four_decimals* has been derived from *xsd*:*decimal* through the enumeration of the four values “-1”, “0.5”, “1”, and “2.4”, and $(o\ pr\ \text{“}1\text{”}\hat{}\hat{}ex{:}four\_decimals) \in K$. Therefore, $(o,\ pr, \text{“}1\text{”}, C^{four\_decimals}(v)) \in F_K$, for all $v \in V^{four\_decimals}$ s.t. $v \leq 0.5$ or $v \geq 2.4$. Consider now a KB K' s.t. $K \rightsquigarrow K'$. Assume that the range of pr in K' is *xsd*:*decimal*. Since *primAncestor*(*four_decimal*) = *decimal*, it follows that $n(\text{“}1\text{”}, four\_decimals, decimal) = \text{“}1\text{”}$. Additionally, $M(0.5, four\_decimals, decimal) = 0.5$ and $m(2.4, four\_decimals, decimal) = 2.4$ Therefore, using rules $R1$ and $R2$, $(o,\ pr, \text{“}1\text{”},\ C^{decimal}(v)) \in F_{K'}$, for all $v \in V^{decimal}$ s.t. $v \leq 0.5$ or $v \geq 2.4$. Additionally, $(o,\ pr, \text{“}1\text{”}, C^{decimal}(v)) \in P_{K'}$, for all $v \in V^{decimal}$ s.t. $0.5 < v < 2.4$. This means that the RDF triple $(o\ pr\ \text{“}1\text{”}\hat{}\hat{}xsd{:}decimal)$ $\in K'$ can be possibly replaced (after the curation process) by any RDF triple $(o\ pr$ $s\hat{}\hat{}xsd{:}decimal)$, where $s \in L^{decimal}$ s.t. $0.5 < V^{decimal}(s) < 2.4$. □

In the following example, we show that the result of Prop. 7 may not hold in the case that $Pr^{XSD}_{K_1} \subset Pr^{XSD}_{K_2}$.

[31] We consider an example namespace *ex* :.

Example 23. Let KBs K_1, K_2, K_3 s.t. $K_1 \rightsquigarrow K_2 \rightsquigarrow K_3$. Assume that (i) $Pr^{XSD}_{K_1} = \emptyset$, (ii) $Pr^{XSD}_{K_2} = \{pr\}$ with $ran_{K_2}(pr) = xsd{:}integer$ and $I^{XSD}_{K_2} = \{(o\ pr\ \text{“2”}\hat{}\hat{}\ xsd{:}integer)\}$, and (iii) $Pr^{XSD}_{K_3} = \{pr\}$ with $ran_{K_3}(pr) = xsd{:}decimal$ and $I^{XSD}_{K_3} = \{(o\ pr\ \text{“2”}\hat{}\hat{}xsd{:}decimal)\}$.

Then, considering the successive step transitions $(F_{K_1}, P_{K_1}) \rightsquigarrow (F'_{K_2}, P'_{K_2}) \rightsquigarrow (F'_{K_3}, P'_{K_3})$, we derive $F'_{K_2} = \{(o, pr, \text{“2”}, s) \mid s \in C^{integer}$ and $s \neq \text{“2”}\}$ and $F'_{K_3} = \{(o, pr, \text{“2”}, s) \mid s \in C^{decimal}, V^{decimal}(s) \leq 1$, and $V^{decimal}(s) \geq 3\}$. In contrast, considering the one step transition $(F_{K_1}, P_{K_1}) \rightsquigarrow (F_{K_3}, P_{K_3})$, we derive $F_{K_3} = \{(o, pr, \text{“2”}, s) \mid s \in C^{decimal}$ and $s \neq \text{“2”}\}$. Thus, $F_{K_3} \neq F'_{K_3}$.

This is because going from KB version K_1 directly to KB version K_3, “2”$\hat{}\hat{}$ *xsd*:*decimal* is considered the most accurate literal value for property *pr* and subject *o*. In other words, in this case, the RDF triple (*o pr* “2”$\hat{}\hat{}$*xsd*:*decimal*) $\in K_3$ satisfies the *MLSA*. In contrast, going to KB version K_3 through KB version K_2, “2”$\hat{}\hat{}$*xsd*:*decimal* may not be the most accurate literal value for property *pr* and subject *o* and the most accurate one may be any of the literals “*x*”$\hat{}\hat{}$*xsd*:*decimal*, where $x \in (1, 3)$. In other words, in this case, the RDF triple (*o pr* “2”$\hat{}\hat{}$*xsd*:*decimal*) $\in K_3$ may not satisfy the *MLSA*. □

Example 24. Consider a KB K and a property $hasColour \in Pr^{XSD}_K$ s.t. $ran_K($ *hasColour*$) =$ *ex*:*six_colours*, which is derived from *xsd*:*string* through the enumeration of the six values “blue”, “green”, “red”, “yellow”, “black”, and “white”. Let $d =$ *string*. Note that *ordered*(*d*) does not hold and $L2C^d(\text{“}red\text{”}) = \text{“}red\text{”}$. Then, *InitPossEnumeration*($K, ($*o hasColour* “*red*”$\hat{}\hat{}$*ex*:*six_colours*))=(*o*, *hasColour*, “*red*”, (“*red*”, *ex*:*six_colours*)). Note that $PossRep_K((o, hasColour, \text{“}red\text{”}, (\text{“}red\text{”},$ *ex*:*six_colours*)) = {(*o*, *hasColour*, “*red*”, “*red*”)}. □

Example 25. Consider KBs K, K' s.t. $K \rightsquigarrow K'$ and a property $hasDate \in Pr^{XSD}_K$ s.t. $ran_K(hasDate) = xsd{:}date$ and $ran_{K'}(hasDate) = xsd{:}dateTime$. Assume that $(o, hasDate, \text{“}1999-03-15\text{”}, \text{“} > 1999-03-14\text{”}, \text{“} < 1999-03-16\text{”}) \in P^{\mathtt{fin}}_K$. Then, $t = (o, hasDate, \text{“}1999-03-15T00{:}00{:}00\text{”}, \text{“} > 1999-03-14T23{:}59{:}59\text{”}, \text{“} < 1999-03-16T00{:}00{:}00\text{”}) \in P^{\mathtt{fin}}_{K'}$. Note that $PossRep_{K'}(t) = \{(o, hasDate, \text{“}1999-03-15T00{:}00{:}00\text{”}, C^d(v)) \mid v \in V^d$ and $1999-03-14T23{:}59{:}59 < v < 1999-03-16T00{:}00{:}00\}$, where $d = dateTime$. □

Example 26. Consider KBs K, K' s.t. $K \rightsquigarrow K'$ and a property $hasGrade \in Pr^{XSD}_K$ s.t. $ran_K(hasGrade) =$ *ex*:*five_integers*, which is derived from *xsd*:*integer* through the enumeration of the five values “0”, “1”, “2”, “3”, and “4”. Additionally, $ran_{K'}($ *hasGrade*$) =$ *ex*:*nine_decimals*, which is derived from *xsd*:*decimal* through the enumeration of the nine values “0”, “0.5”, “1”, “1.5”, “2”, “2.5”, “3”, “3.5”, and “4”. Assume that $(o, hasGrade, \text{“2”}, \text{“} > 1\text{”}, \text{“} < 3\text{”}) \in P^{\mathtt{fin}}_K$. Then, $t = (o, hasGrade, \text{“2”}, \text{“} > 1\text{”}, \text{“} < 3\text{”}) \in P^{\mathtt{fin}}_{K'}$. Note that $PossRep_{K'}(t) = \{(o, hasGrade, \text{“2”}, C^d(v)) \mid v \in V^d$ and $1 < v < 3\}$, where $d =$ *nine_decimals*. □

Example 27. Consider KBs K, K' s.t. $K \rightsquigarrow K'$ and a property $hasDate \in Pr^{XSD}_{K'} \setminus Pr^{XSD}_{K'}$ s.t. $ran_{K'}(hasDate) = xsd{:}date$ (i.e. *pr* is a new XSD-valued property of K'). Assume that (*o hasDate* “1999−03−15”$\hat{}\hat{}$ *xsd*:*date*) $\in K'$. Then, *InitDatatype*($K, ($*o hasDate* “1999 − 03 − 15”$\hat{}\hat{}$*xsd*:*date*)) = $(o, hasDate, \text{“}1999-03-15\text{”}, \text{“} > 1999-03-14\text{”}, \text{“} < 1999-03-16\text{”}) \in P^{\mathtt{fin}}_{K'}$. Note that $PossRep_{K'}((o, hasDate, \text{“}1999-03-15\text{”}, \text{“} > 1999-03-14\text{”}, \text{“} < 1999-03-16\text{”})) = \{(o, hasDate, \text{“}1999-03-15\text{”}, \text{“}1999-03-15\text{”})\}$. □