Support uniforme de types de données personnalisés dans RDF et SPARQL

Maxime Lefrançois, Antoine Zimmermann

Univ Lyon, MINES Saint-Étienne, CNRS, Laboratoire Hubert Curien UMR 5516,
F-42023 Saint-Étienne, France
prenom.nom@emse.fr

Résumé. Les littéraux sont les nœuds terminaux du modèle de données RDF, et permettent d'encoder des données telles que des nombres (`"12.5"^^xsd:decimal`), des dates (`"2017-01-26T23:57:15"^^xsd:dateTime`), ou tout autre type d'information (`"vert pomme"^^ex:couleur`). Les moteurs RDF/SPARQL savent tester l'égalité ou comparer les littéraux RDF dont le type de données leur est connu (ce qui est le cas de `xsd:decimal` et `xsd:dateTime`). Mais lorsqu'un type de données est inconnu d'un moteur RDF/SPARQL (comme `ex:couleur`), il n'a à priori aucun moyen d'en « découvrir » la sémantique. Dans cet article, nous attaquons ce problème et étudions comment permettre: (i) aux éditeurs de données de publier la définition de types de données personnalisés sur le Web, et (ii) aux moteurs RDF/SPARQL de découvrir à la volée ces types de données personnalisés, et de les utiliser de manière uniforme. Nous discutons de différentes solutions possibles qui tirent partie des principes du Web des données, et détaillons une solution concrète basée sur le déréférencement et le langage JavaScript, suffisemment générique pour être utilisée pour des types de données personnalisés arbitrairement complexes.

1 Introduction

Un littéral RDF est composé d'une chaîne de caractères UNICODE (la forme lexicale) et d'une IRI de type de données [1] qui identifie un type de données (Cyganiak et al., 2014, §5). Ils forment avec les IRI et les nœuds anonymes les atomes du modèle de données RDF. Les IRI forment les liens qui tissent la toile du Web des données, mais ce sont les littéraux qui *in fine* encodent les données telles que des nombres (`"12.5"^^xsd:decimal`), des dates (`"2017-01-26T23:57:15"^^xsd:dateTime`), ou tout autre type d'information (`"vert pomme"^^ex:couleur`). Les moteurs RDF/SPARQL savent tester l'égalité ou comparer les littéraux RDF dont le type de données leur est connu (ce qui est le cas de `xsd:decimal` et `xsd:dateTime`). On dit alors qu'ils *supporte* le type de données. En pratique, les moteurs sont programmés pour supporter un ensemble fini de types de données. [2]

1. Et lorsque l'IRI du type de données est `rdf:langString`, d'une étiquette de langue. Cependant ce type de littéral ne nous intéresse pas dans le cadre de cette étude.

2. Il s'agit au moins de l'ensemble des types de données XSD, mais certains moteurs en supportent d'autres comme ceux du standard OGC GeoSPARQL (Perry et Herring, 2012).

Cependant lorsqu'un type de données est inconnu d'un moteur RDF/SPARQL (comme `ex:couleur`), il n'a à priori aucun moyen d'en « découvrir » la sémantique. Cet article, qui est un résumé de Lefrançois et Zimermann (2016), propose une solution à ce problème. Il serait en effet intéressant de pouvoir ainsi rendre plus flexible la puissance descriptive du modèle de données RDF, car les types de données personnalisés permettent des descriptions plus concises pour certaines structures. Par exemple la description du diamètre moyen de la Terre avec un type de données personnalisé `cdt:length` nécessiterait un seul triplet :

```
@Prefix dbr: <http://dbpedia.org/resource/> .
@prefix cdt: <http://w3id.org/lindt/custom_datatypes#>.
+
dbr:Earth <http://dbpedia.org/ontology/Planet/meanRadius> "6371.0 km"^^cdt:length .
```

Alors que la même longueur décrite avec l'ontologie QUDT (Hodgson et al., 2014) en nécessite quatre, ce qui alourdit le stockage et rend complexe les requêtes sur les longueurs :

```
@Prefix qudt: <http://qudt.org/schema/qudt#> .
@Prefix qudt-unit: <http://qudt.org/vocab/unit#> .

dbr:Earth <http://dbpedia.org/ontology/Planet/meanRadius> _:quantity .
_:quantity qudt:quantityValue _:value .
_:value qudt:numericValue "6371.0"^^xsd:double .
_:value qudt:unit qudt-unit:kilometre .
```

Nous souhaitons donc plus précisément étudier comment permettre : (i) aux éditeurs de données de publier la définition de types de données personnalisés sur le Web (par exemple `cdt:length`), et (ii) aux moteurs RDF ou SPARQL de découvrir à la volée ces types de données personnalisés, et de les utiliser de manière uniforme. La suite de cet article est organisé comme suit. La section 2 identifie les besoins pour arriver à ce résultat, et étudie quelques options d'implémentation possibles. La section 3 introduit alors une solution qui adresse spécifiquement le cas des types de données arbitrairement complexes, décrit son implémentation, et résume les résultats d'une expérimentation sur un jeu de données réel.

2 Support à la volée des types de données

Nous souhaitons donc que les moteurs RDF ou SPARQL puissent découvrir à la volée un type de données à partir de l'IRI qui l'identifie, puis traiter les littéraux ayant ce type de données de manière uniforme. Cette section donne un aperçu de ce qui est nécessaire, et étudie différentes options possibles.

Dans cet article, un littéral se compose d'une chaîne UNICODE appelée *forme lexicale*, et d'une IRI appelée *IRI de type de données*.[3] Nous identifions une IRI arbitraire par `a`, `b`, etc., et une chaîne UNICODE par `s`, `t`, etc. La recommandation RDF 1.1 définit un type de données D comme une structure comprenant : (i) un ensemble $L(D)$ de chaînes UNICODE, appelé l'*espace lexical* ; (2) un ensemble $V(D)$, appelé l'*ensemble de valeurs* de D ; (3) une application $L2V(D) : L(D) \rightarrow V(D)$, appelée *lexical-to-value mapping*, qui associe à toute chaîne de $L(D)$ une valeur dans $V(D)$. Nous utilisons les définitions de Hayes et Patel-Schneider (2014), notamment la notion de *IRI reconnue*. Lorsqu'un un moteur RDF ou SPARQL reconnaît une IRI qui identifie un type de données D_a, on dit qu'il *supporte* D_a.

3. Nous ne considérons pas les littéraux avec une étiquette de langue.

Fonctionnalités nécessaires pour le raisonnement et le requêtage. Par définition, le type de données identifié par une IRI spécifie la valeur que la forme lexicale a pour ce type. C'est une structure mathématique qui ne peut pas toujours être représentée dans un format interprétable par l'ordinateur. Il n'est pas nécessaire pour le moteur RDF ou SPARQL de "comprendre" la structure mathématique du type de données à reconnaître. Il lui suffit d'implémenter certaines opérations qui peuvent elles être programmées. Nous identifions trois fonctionnalités nécessaires et suffisantes. Un moteur RDF qui supporte un type de données D_a identifié par une IRI `a` doit seulement être capable de vérifier la bonne forme d'un littéral de type de données `a`, ou l'égalité de deux tels littéraux. Ces fonctionnalités suffisent également à un moteur SPARQL pour déterminer la correspondance entre graphes simples. La seule fonctionnalité additionnelle nécessaire pour SPARQL est de pouvoir ordonner certains littéraux, lorsque le type de données s'y prête.

Bonne forme Etant donné une chaîne UNICODE `s`, la forme lexicale `s` est-elle bien formée dans D_a, i.e., Appartient-elle à l'espace lexical de D_a ? De manière équivalente, le littéral `"s"^^a` est-il bien typé ? i.e., $\mathtt{s} \in L(D_a)$.
Par exemple, `"12.5"` est bien formé dans `xsd:decimal`, alors que `"abc"` ne l'est pas (`"12.5"^^xsd:decimal` est bien typé, et pas `"abc"^^xsd:decimal`).

Egalité Etant données deux chaînes UNICODE `s`,`t`, les littéraux `"s"^^a` et `"t"^^a` ont-ils la même valeur ? i.e., $L2V(D_a)(\mathtt{s}) = L2V(D_a)(\mathtt{t})$.
Par exemple, `"0.50"^^xsd:decimal` et `".5"^^xsd:decimal` ont la même valeur.

Comparaison Etant données deux chaînes UNICODE `s`,`t`, la valeur de `"s"^^a` est-elle plus petite (resp., grande) que celle de `"t"^^a` ? i.e., $L2V(D_a)(\mathtt{s}) < L2V(D_a)(\mathtt{t})$
(resp., $L2V(D_a)(\mathtt{s}) > L2V(D_a)(\mathtt{t})$).

Ces fonctionnalités sont suffisantes pour vérifier la D-implication simple entre graphes RDF, c'est à dire l'implication simple en reconnaissant un ensemble de types de données D Hayes et Patel-Schneider (2014, §7). Lefrançois et Zimermann (2016) discutent des cas plus compliqués, et étendent ces définitions à la reconnaissance d'un ensemble quelconque de types de données, dont les espaces de valeurs peuvent se chevaucher.

Options possibles d'implémentation. Les moteurs RDF qui n'ont pas d'implémentation codée en dur pour une IRI de type de données doivent pouvoir obtenir une version calculable des fonctions décrites à la section §2. Ceci peut être impossible pour certains types de données lorsque le problème associé est indécidable. Par exemple pour un type de données qui encode les formules en logique du 1er ordre, et qui aurait pour espace de valeur l'ensemble des classes d'équivalence de formules vis-à-vis de l'implication en logique du 1er ordre. Dans cet article, nous nous intéressons aux cas pratiques pour lesquels la bonne forme, l'égalité, et la comparaison sont des fonctions calculables.

Toute solution pratique nécessite un accord entre l'éditeur[4] et le consommateur sur le mécanisme à utiliser pour présenter et exploiter les fonctionnalités requises. Ces fonctions pourraient être fournies par un service centralisé d'enregistrement de types de données, où les éditeurs enregistrent la spécification pour leur type de données. Une telle solution est mal pratique et en désaccord avec les principes fondamentaux du Web.

4. L'éditeur est celui qui spécifie le type de données identifié à une IRI.

Dans la suite de cet article, nous nous focalisons donc sur les solutions qui partent du principe que la version calculable des fonctionnalités est accessible en déréférençant l'IRI du type de données. Ce principe est justement suggéré dans la section 7 de la recommandation RDF 1.1 Semantics. Nous nous restreignons aux IRI HTTP.

Librairies spécifiques aux moteurs. Les moteurs ARQ et SESAME permettent d'enregistrer des classes qui vérifient la bonne forme et l'égalité pour un type de donnée. Il permettent également d'enregistrer des classes qui implémentent des fonctions de filtre SPARQL, et permettraient la comparaison pour un type de données. Le moteur pourrait accéder à une archive avec les classes nécessaires à l'IRI à reconnaître. Bien que cette solution soit simple, elle est dépendante des implémentations, et l'éditeur du type de données aurait à écrire des classes pour chaque moteur RDF. Elle présente également une menace importantes de sécurité, tout du moins en Java.

Langage de script. Au lieu d'utiliser des classes compilées spécifiques à chaque implémentation, il s'agit de fournir le code des fonctions nécessaires dans un langage de script. L'utilisation de ces fonctions devrait alors être implémentée une seule fois pour chaque moteur. Un bon candidat pour ce langage de script est JavaScript, pour qui des environnements d'exécution sécurisés existent dans beaucoup de langages de programmation. Cette solution permet d'utiliser l'expressivité d'un langage de programmation, et permet la spécification de types de données arbitrairement complexes.

Service Web. Une approche alternative consiste à rendre l'exécution de ces fonctions accessibles via un service Web. Bien que cette solution nécessiterait une haute accessibilité du service, nous souhaiterions l'étudier à l'avenir.

Description basée sur une ontologie. Pour beaucoup de types de données simples, il semble excessif d'utiliser l'expressivité complète d'un langage de programmation. Il serait intéressant d'utiliser une ontologie pour représenter ces cas simples, possiblement inspiré des restrictions de types de données OWL 2 (W3C OWL Working Group, 2012). Nous investiguons actuellement cette solution.

3 Types de données arbitraires spécifiés par des scripts

Nous proposons une première solution concrète basée sur le langage JavaScript, son implémentation, et son évaluation.

Directives pour l'éditeur. Il s'agit de : (1) utiliser une IRI HTTP `a` pour identifier un type de données D_a ; (2) exposer au moins une représentation en JavaScript pour le type de données à cette URL ; (3) le script doit implémenter une interface simple `CustomDatatypeFactory` dont l'unique méthode permet d'obtenir des objets qui, eux, implémentent une interface `CustomDatatype` [5]. Ces interfaces et l'ensemble des contraintes formelles qu'une implémentation doit respecter sont décrites sur le site web `http://w3id.org/lindt/spec.html`.

5. Cette indirection est nécessaire car plusieurs types de données peuvent être définis dans le même document, comme `xsd:string` et `xsd:int` sont définis dans le document à l'URL `http://www.w3.org/2001/XMLSchema`. Il faut donc que le moteur RDF sache quelle partie du code exécuter pour chaque type de données.

Directives pour le moteur RDF ou SPARQL. Lorsqu'un moteur RDF ou SPARQL rencontre un littéral avec un type de données inconnu D_a identifié par une IRI HTTP `a` : (1) il opère un HTTP GET à `a` avec l'option HTTP `Accept: application/javascript` ; (2) si l'opération est un succès, il interprète le script, exécute la fonction `getDatatype(a)` et récupère un objet `da` qui contient la version calculable des fonctionnalités nécessaires pour supporter le type de données D_a ; (3) il utilise les fonctions de `da` pour valider une forme lexicale, vérifier l'égalité de deux littéraux, ou comparer deux littéraux ayant le type de données D_a.

Publication d'un type de données. Nous avons publié un premier type de données d'IRI `http://w3id.org/lindt/v1/custom_datatypes#length`, abrégée `cdt:length`, pour décrire des mesures de longueurs. Notre serveur utilise la négociation de contenu pour servir la spécification de `cdt:length` : en JavaScript, en Turtle, ou en HTML. L'espace lexical est la concaténation d'un `xsd:double`, un espace optionnel, et d'une unité. Les littéraux suivants sont tous bien typés et ont la même valeur.

```
"1 mile"^^cdt:length          "63360 inches"^^cdt:length    "1609.344 metre"^^cdt:length
"5280 ft"^^cdt:length         "1.609344km"^^cdt:length      "1.609344E+6 mm"^^cdt:length
```

Implémentation du support à la volée dans Jena et ARQ. Nous avons implémenté les directives dans les moteurs Jena et ARQ [6]. Les modifications au cœur de Jena et ARQ qui ont été nécessaires sont précisées par Lefrançois et Zimermann (2016).

Expérimentation. L'article complet Lefrançois et Zimermann (2016) décrit notre expérimentation sur 223,768 triplets de DBpedia 2014 qui décrivent des longueurs. L'objectif est d'évaluer le temps de chargement d'un jeu de données ainsi que le temps d'exécution de requêtes correspondantes pour différentes fractions de :

(a) les triplets de DBpedia qui utilisent des types de données personnalisés ;

(b) leurs équivalents décrit avec l'ontologie QUDT [7] (4 fois plus de triplets) ;

(c) leurs équivalents décrit avec notre type de données `cdt:length` découvrable à la volée.

Le site web `http://w3id.org/lindt/spec.html#h-experiments` décrit les requêtes exécutées, et contient le matériel pour reproduire les expérimentations ainsi que les résultats bruts. Ces résultats montrent que le temps de chargement des jeux de données (c) sont très proches de ceux de (b), avec une pénalité moyenne de 468 ms pour la découverte du type de données. D'un autre côté, les jeux de données (c) ont les meilleures performances en ce qui concerne le temps d'évaluation des requêtes, qui sont par ailleurs plus concises.

4 Conclusion

Les types de données personnalisés sont peu utilisés sur le Web des données car ils ne facilitent pas l'interopérabilité. Si ils pouvaient être supportés de manière générique, cela faciliterait la publication de jeux de données de domaines qu'il peut être difficile de représenter avec des types de données standards. Nous avons proposé des principes qui permettraient :

6. `https://github.com/thesmartenergy/jena`.
7. http ://qudt.org/

(i) aux éditeurs de données de publier la définition de types de données personnalisés sur le Web, et (ii) aux moteurs RDF ou les moteurs de requête SPARQL de découvrir à la volée ces types de données personnalisés, puis de les utiliser pour accomplir des opérations de manière uniforme. Nous avons proposé une première implémentation de ces principes pour démontrer leur applicabilité, et résumé une évaluation de notre approche sur un jeu de données données réelles de DBpedia. Certaines directions possibles de recherches ont été mentionnées dans le corps de cet article. Nous souhaitons également développer une librairie de types de données pour initier l'adoption notre approche à plus grande échelle.

Remerciements

Ce travail a été partiellement financé par le projet ITEA2 12004 Smart Energy Aware Systems (SEAS), le projet ANR 14-CE24-0029 OpenSensingCity, et une convention bilatérale de recherche avec ENGIE R&D.

Références

Cyganiak, R., D. Wood, et M. Lanthaler (2014). RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation 25 February 2014. W3C Recommendation, W3C.

Hayes, P. et P. F. Patel-Schneider (2014). RDF 1.1 Semantics, W3C Recommendation 25 February 2014. W3C Recommendation, W3C.

Hodgson, R., P. J. Keller, J. Hodges, et J. Spivak (2014). QUDT - Quantities, Units, Dimensions and Data Types Ontologies . Technical report, NASA.

Lefrançois, M. et A. Zimermann (2016). Supporting Arbitrary Custom Datatypes in RDF and SPARQL. In *Proceedings of the Extended Semantic Web Conference, ESWC*.

Perry, M. et J. Herring (2012). OGC GeoSPARQL - A Geographic Query Language for RDF Data. Ogc implementation standard, Open Geospatial Consortium.

W3C OWL Working Group (2012). OWL 2 Web Ontology Language Document Overview (Second Edition), W3C Recommendation 11 December 2012. Technical report, W3C.

Summary

Literals are terminal nodes for the RDF data model, where is encoded actual data such as decimals (`"12.5"^^xsd:decimal`), dates (`"2017-01-26"^^xsd:date`), or any other information (`"apple green"^^ex:color`). RDF and SPARQL engines can test equality or compare literals whose datatype they know (which is the case for `xsd:decimal` and `xsd:date`). But when a datatype is unknown (like `ex:color`), then there is no direct means to "discover" its semantics. This paper tackles this problem and show how : (i) data publishers can publish the definition of arbitrary custom datatypes on the Web, and (ii) generic RDF or SPARQL engines can discover them on-the-fly, and perform operations uniformly. We discuss different possible solutions that leverage the Web of Data principles, and describe a simple one based on dereferencing and JavaScript, that can be used for arbitrarily complex custom datatypes.