Incompleteness-aware Programming with RDF Data

Stefan Scheglmann  Gerd Groener  Steffen Staab  Ralf Lämmel
University of Koblenz-Landau
{schegi, groener, staab, laemmel}@uni-koblenz.de

Categories and Subject Descriptors D.3.2 [Language classifications]: Multiparadigm languages; D.3.3 [Language constructs and features]: Data types and structures; E.1 [Data structures]: Distributed data structures

General Terms Languages, Design, Theory

Keywords Semantic Data, Incomplete Data, Linked Open Data, Data Access, Information-rich Programming, Structural Typing, Nominal Typing

1. Introduction

It is becoming common to publish data on the Web by using semantic technologies like RDF as a flexible formalism for structured data representation. In particular, industrial and governmental authorities have started to publish open data in RDF; see, for example, the governmental data collection project1 and Wikidata2 (i.e., the semantic data representation of Wikipedia).

The publication and integration of RDF data relies on the Linked Data principles [7], which describe simple, best practices for using RDF, identifiers (URIs), and protocols (HTTP) for exposing, sharing, connecting, and consuming data on the Web.

However, access to RDF data within programs is difficult since (i) RDF offers quite flexible means for data representation, (ii) schemas are often missing, (iii) type information is often missing in actual data; (iv) data on the Web evolves regularly, and (v) data might be incomplete due to different vocabularies and data sources that are used simultaneously. Thus, programming on semantic data on the Web means to access continuously changing, distributed data sources without a strict schema.

In this paper, we initiate work towards a programming model that ultimately will address the aforementioned challenges. We believe that a cornerstone of such a programming model may be a property-based data-access approach, which would complement the more classical, nominal type-based approach. (Such a view was briefly sketched in [17].) The impact of a property-based view on programming model and programming-language support is systematically discussed in the rest of the paper.

1. Motivating Example

While the flexibility of RDF supports its widespread adoption, it is quite difficult to realize robust access to RDF data in programs. The relevant issues will be demonstrated with an example.

Assume an e-participation platform where citizens can actively participate in political discussions and vote for decisions. For convenience, citizens can register in this platform with their existing accounts in social networks like Facebook or LinkedIn. As a consequence, (1) the data of the citizens may be incomplete (e.g., the date of birth may be missing), (ii) properties may have different names, (iii) type statements may be missing (e.g., a person may lack an explicit tag for a citizen type), and yet other data variations are conceivable. Thus, the e-participation platform cannot assume a strict schema for the properties of citizens.

Consider the following RDF statements about some citizens.

```rml
/* Explicit association with the citizen type */
alice rdfs:type rdfs:Person.
bob rdfs:type rdfs:Person.
dave rdfs:type rdfs:Person.

/* Citizen data for Alice */
alice rdfs:label "Alice".
alice foaf:mbox "alice@home.com".

/* Citizen data for Bob */
bob rdfs:label "Bob".
bob foaf:mbox "bob@mail.com".

/* Candy has name and mbox, but no citizen type. */
candy foaf:name "Candy".
candy foaf:mbox "candy@coldmail.com".

/* Dave has no interest. */
dave foaf:name "Dave".
dave foaf:mbox "dave@online.com".
```

The remainder of the paper is structured as follows. In Section 2, we explain the aforementioned challenges in more detail with the help of a motivating example. In Section 3, we analyze requirements for a suitable programming model and programming-language support for property-based data access. In Section 4, we present some related research that may be helpful in continuing work on the programming model. The paper is concluded in Section 5.

1. Motivating Example

While the flexibility of RDF supports its widespread adoption, it is quite difficult to realize robust access to RDF data in programs. The relevant issues will be demonstrated with an example.

Assume an e-participation platform where citizens can actively participate in political discussions and vote for decisions. For convenience, citizens can register in this platform with their existing accounts in social networks like Facebook or LinkedIn. As a consequence, (1) the data of the citizens may be incomplete (e.g., the date of birth may be missing), (ii) properties may have different names, (iii) type statements may be missing (e.g., a person may lack an explicit tag for a citizen type), and yet other data variations are conceivable. Thus, the e-participation platform cannot assume a strict schema for the properties of citizens.

Consider the following RDF statements about some citizens.

```rml
/* Explicit association with the citizen type */
alice rdfs:type rdfs:Person.
bob rdfs:type rdfs:Person.
dave rdfs:type rdfs:Person.

/* Citizen data for Alice */
alice rdfs:label "Alice".
alice foaf:mbox "alice@home.com".

/* Citizen data for Bob */
bob rdfs:label "Bob".
bob foaf:mbox "bob@mail.com".

/* Candy has name and mbox, but no citizen type. */
candy foaf:name "Candy".
candy foaf:mbox "candy@coldmail.com".

/* Dave has no interest. */
dave foaf:name "Dave".
dave foaf:mbox "dave@online.com".
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Data Driven Functional Programming Workshop 2013 January 22, 2013, Rome, Italy
Copyright © 2013 ACM [to be supplied]... $15.00
The present paper explores the idea of complementing the common, nominal typing approach, as in a class- and interface-based type system in OO programming, with a more structural typing or property-based approach. Programmers should hence be provided with additional features for expressing type constraints. For instance, it should be possible to define a method not for a nominal (class) type, but for a constraint that takes the form of a set of properties, which an object has to provide to be accessed with the method.

3 The F# Type Provider http://msdn.microsoft.com/en-us/library/hh156509.aspx, last visit 9th November 2012

3. Property-based Data Access

Resources in RDF can be classified in a twofold way: (1) explicitly, by providing a type or (2) implicitly, via the definition of its properties. In [6], we investigated these resource characteristics on extracted schematic information from the Billion Triples Challenge 2011 dataset. These investigations have shown that the correlation between the type and the properties attached to a resource is relatively high. Also, the properties of a resource were found to be relatively indicative for the type.

When applied to the motivating example, we might reason that Candy is effectively also of type ex:citizen because the resource has properties foaf:name and foaf:mbox—just like the known citizens.

Accordingly, we submit the following hypothesis:

**Hypothesis** Property-based data access is well suited to classify, query, transform, construct, and generally process RDF data in programs for information-rich applications.

While the hypothesis obviously calls for more validation, we will focus here on issues of programming model and programming-language support, as they have to be addressed ultimately, if property-based data access is to be supported.

**Type Hierarchies** Nominal type hierarchies play an established role in programming, especially according to the OO programming paradigm. Thus, we need to understand how to realize a type hierarchy in a programming model for RDF data. In particular, we need to understand the implications of a property-based approach. These are more elementary questions:

- How to build a useful and robust type hierarchy on top of type information that is essentially based on property sets? In principle, we may use a formal concept analysis to construct a type hierarchy, i.e., a lattice including all exercised property sets. In the motivating example, all resources (but the group) carry a foaf:name property—this singleton set thus represents one node in the lattice; another node is the set \{foaf:name, foaf:mbox, foaf:interest\}, as exercised by both Bob and Alice.
- How to treat the rdf:type statements and the corresponding type hierarchy? Should these types be ignored or should they coexist with property set-based types? In the latter case, should both type hierarchies be merged or should they be kept separate?
- How to name the types in the new type hierarchy for property sets? For instance, how to name the different property sets for citizen data? We may be able to reuse names from existing type statements for some of the exercised property sets. Other property sets will require additional names.

**Incompleteness of Data** As mentioned earlier, incomplete RDF data is quite common, especially in the case of Linked Open Data. Various issues are implied by incompleteness. Overall, how should a program deal with incomplete data? These are more elementary questions:

- To what extent can we cater for incompleteness at a global scope, i.e., at the level of declaring the nominal and property set-based types in a program, versus the local scope of specifically expressing prerequisites of methods?
- How to express that certain properties are required and others are optional within the scope of some functionality, as implemented in a method? We can think of the corresponding declarations as some sort of contracts. A method must not be applied to a resource that lacks properties required by the method. When composing functionality, such contracts may need to be synthesized appropriately (automatically).
What kinds of exception handling or defaults (think of default values for missing properties) are needed? If data access is shielded by simple, aggressive exception handlers or defaults, then the corresponding functionality may behave incorrectly. A more advanced approach should leverage the aforementioned contracts for required and optional properties. Hence, we like to think of active contracts because they would actually affect (i.e., select) program behavior.

What about missing rdf:type statements? In principle, we may try to derive these statements. We may also consider missing type statements as a variation on missing properties (see above), but there is the important characteristic of types to engage in a hierarchy and hence, missing types must be considered differently.

How to become aware and how to respond to evolving data? Evolution may clearly affect incompleteness. Without precautions in the program, data access functionality may break with or without the programmer noticing. The ultimate programming environment should monitor evolving data sources, e.g., in terms of the active contracts in existing programs, so that the programmer can be notified of data issues or programs could become more adaptive.

**Active Contracts** Without a claim of completeness, we think of the following forms of contracts:

- A property is **required** for an argument of a method.
- A property is **optional** for an argument of a method.
- Such declarations also make sense for **results** of methods, in which case a required property can be assumed to be definitely present in the result and an optional property is expected but not required.
- Some regime of open- versus closed-world assumption may be selected, with rich implications. For instance, if an open-world assumption is assumed, then it may be acceptable that some potential properties are not pointed out in contracts.
- We may also use (nominal) **types** in contracts in a manner similar to properties.
- Logical connectors (negation, conjunction, disjunction, etc.) can be used to compose contracts. For instance, this would allow for exclusion contracts on specific properties so that we can declare which kind of property an object must not have to be accepted by a method.
- We may also want to constrain the number of values in a property. For instance, one could state that a resource has to have at least one foaf:interest to be of type ex: citizen.

**Auto typing** Each resource has a purely structural type: its property set (possibly also taking into account the data types for the properties). Additionally, a resource may have any number of declared nominal types based on explicit rdf:type statements. (As we discussed earlier, such nominal types are only available, if the programming model is designed to preserve rdf:type statements and the associated type hierarchy.)

In the view of incompleteness and the property set-based type hierarchy, we may want to associate resources with additional types. We refer to this process as **auto typing** and we assume that it is supported by the type system at runtime. Thus, how to define and realize auto typing? These are more elementary questions:

- Does auto typing attach rdf:type-like types to resources or do we assume that such types are only declared explicitly by the rdf:type statements in data sources? In the motivating example, auto typing could associate the ex: citizen type with any resource that has, for example, the properties foaf: name and foaf: mbox. Interestingly, if such auto typing is considered, then data access can increasingly rely on rdf:type-based as opposed to property-based active contracts.
- What sorts of auto-typing rules are needed?
- More specifically, how to attach additional rdf:type statements to resources? It is reasonable to assume that conditions of auto-typing rules take the form of property sets. Thus, nominal types become essentially tests for property sets.
- Can we achieve some auto-typing such that the characteristic property sets for types are inferred rather than declared? As we noted earlier, schema extraction approaches are focused on design time and snapshots of data. Instead, we may try to maintain characteristic property sets of rdf: types (i.e., the intersection of all property sets for actual resources of each type) along runtime and use it to auto-type resources with the same property set. This approach is challenging in so far that types of resources would evolve along with the exercised data at runtime.
- If we assume that names are associated with the nodes of the property set-based lattice (see the discussion of type hierarchies above), then we clearly require auto-typing rules to establish those nominal types.
- Does auto typing cover retyping of resources at runtime? Consider a program which constructs a new resource and eventually and possibly in separate method adds properties for name and mail address. Should the corresponding resource be associated with the ex: citizen type, i.e., retyped? How to achieve such retyping? How does retyping affect the programming model? How to avoid that the programmer exercises property sets (and property types) that may not be intended?

A discussion similar to auto typing would also be needed for the issue of implicit/derived properties, as indicated earlier, but it is skipped here for brevity.

4. **Related Work**

Future work towards a programming model and language support for incompleteness-aware programming with RDF data should be informed by diverse programming language research. We list some pointers in this section.

One essential idea is to liberate programming from the dominant nominal, hierarchically typed approach by including structural typing expressiveness, where necessities of functionality can be expressed in a property-based as opposed to class- or interface-based manner—similar to structural typing approaches in an OO-language setting [4] or extensible record systems studied in different language settings [2, 3, 9]. In the context of Linked Open Data, we face the additional challenge of discovering, managing, and comprehending property sets including their continuous evolution in programs.

Another essential idea is to support robustness or adaptivity in programming on “variable data” so that functionality can be composed in a way that applicability conditions are explicitly expressed and generic strategies take care of locating data of interest and defaulting for other data—similar to generic functional programming [11] or adaptive object-oriented programming [14] and related approaches [1, 12]. For example, such strategies would allow us to process containers with diverse resources such that the basic blocks of behavior are applied only to resources that meet specific property set-based expectations. Advanced approaches to exception
handling [18] seem to be relevant here as well. Also, constructs on nullable types [15] and type case are relevant [5].

Yet another essential idea is to enable more dynamicity of data in terms of its associated types, as the type of a resource may change along its evolution or due to additional type information becoming available as more and more resources are processed from the input. In this context, we observe related work dealing with dynamic contracts such that objects may associate contracts over life time or enter relationships or contexts of role-based collaboration. Ultimately, we seek a programming model such that instances can be created in a manner such that assumed types do not need to be expressed explicitly, but applicable types are instead inferred or checked on the grounds of statically as well as dynamically available premises. We believe that some research on dynamically associated contracts [13] as well as roles and context in programming models and programming-language support for accessing and processing incomplete RDF data.

5. Conclusion

In this paper, we have hypothesized that “property-based data access is well suited to classify, query, transform, construct, and generally process RDF data in programs for information-rich applications”. A property-based approach is needed to cope with missing types, missing properties, and data integration for Linked Data sources. We have discussed the challenges and open questions around such a property-based approach. This discussion may be helpful in conducting future research on programming models and programming-language support for accessing and processing incomplete RDF data.

Acknowledgments

This work has been supported by Microsoft.

References


