Reified Literals: A Best Practice Candidate Design Pattern for Increased Expressivity in the Intelligence Community

Eric Peterson,

Global Infotek, 1920 Association Drive
Reston, VA 20191, USA
epeterson@globalinfotek.com

Abstract. Reifying literals clearly increases expressivity. But reified literals appear to waste memory, slow queries, and complicate graph-based models. We show where this practice can be comparable to unreified literals in these respects and we characterize the cost where it is not. We offer examples of how reification allows literals to participate in a variety of relations enabling a marked increase in expressivity. We begin with a case study in reified person names, and then extend this analysis to reified dates and simple reified scalar values. We show benefits for name matching and temporal analysis such as would be of interest to the Intelligence Community (IC). We then show how these same sorts of analyses can drive or inform any decision as to whether to reify literals.

Keywords: reified literal, semantic, ontology, expressivity, best practice, design pattern, Intelligence Community

1 Introduction

Reifying literals is not uncommon among popular ontologies and relational data models. But data architecture teams in the IC can draw from varied backgrounds and the use of reified literals may not be desired.

The practice may not be desired because it appears to waste memory, slow queries, and complicate graph-based models despite the increase in expressivity that it offers. We lay out simple, general metrics for judging such memory waste, slowness and complication. We show where the practice can be comparable in those respects to unreified literals and we characterize the modest cost where it is not. We show how reification allows literals to participate in a variety of relations which foster expressivity. Beginning with a case study comparing reified with unreified person names, we then extend the analysis to dates and simple scalar values. We then show how these same analyses can drive or inform any decision regarding whether to use or not use literal reification. This paper works toward establishing the reified literal design pattern as a best practice component.

We define reified literals as instances that represent literal values. A reification of a name string literal value might be an instance of type *Name* with a datatype property containing the value of the name string. This name instance might then be attached to a person instance by the *givenName* object property statement (See Figure 1).

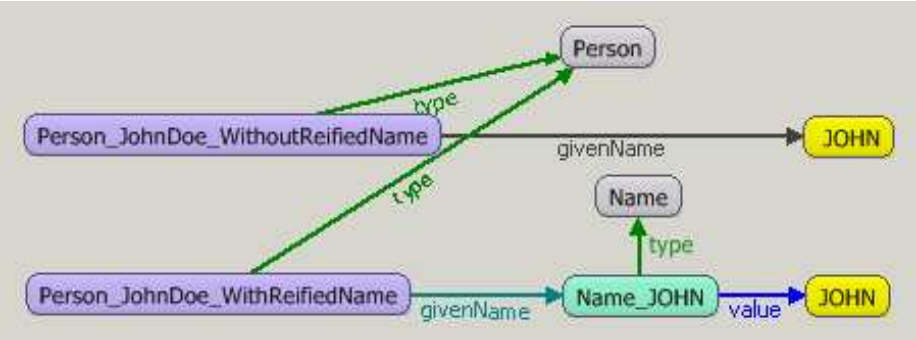

Figure 1: In the top unreified example, *givenName* is a datatype property statement. In the bottom reified example, *givenName* is an object property statement referencing a *Name* instance.

2 Current Practice, Related Work, and Contributions

Literal reification is not an uncommon practice. OpenCyc[1], Iode[2], and SUMO[3], ontologies have pervasively reified literals. DOLCE[4] leaves literal definition to extending ontologies.

W3C Semantic Web Best Practices and Deployment Working Group presented a draft by Hobbs and Pan[5] of a time ontology that uses only reified time literals. Project NeOn's Ontology Design Pattern Repository[6] contains a pattern for reified lexical items (terms). Many more main-stream examples exist.

The closest work related to the reified literals design pattern is Presutti and Gangemi's[7] content ontology design pattern requirements - to which reified literals comply[1].

Reified literals are not new. But we choose to characterize the virtues and cost of using this design pattern. We address some common misconceptions about this design pattern's performance by detailing memory footprint cost, speed, and design

[1] The reified literal design pattern is *computational* in that it is language independent and is encoded in a higher order language (OWL). It is clearly *small*. It is *autonomous* (deployable as a single file). It is *hierarchical* in that the class *Literal* must be subclassed for each particular literal. It is *inference-enabling* in that its reified instances become the foci of relationships that *say something* about the literal. Dates participate in Allen's interval calculus relations via transitive closure for example. The pattern is *cognitively relevant* in that it is intuitive, compact, and captures relevant notions in a domain. It is *linguistically relevant* in that we speak of names, dates, etc. as real things.

complexity. We give examples of the pattern's increase in expressivity. We show how to apply theses analyses to all literals.

3 Methods and Metrics

We describe simple, simple metrics to compare reified literal costs in memory and query speed with respect to those of unreified literals. With the relative complexity of reified literals and the benefits of their increased expressivity, however, we do not attempt to go beyond a qualitative description.

For memory usage comparison the two approaches differ structurally only by the type of one statement (after shared structure is amortized away). With the reified approach the type of the statement in question is *ObjectProperty* and with the un-reified approach it is of type *DatatypeProperty*. If datatype property statements are less compact in a particular implementation than object property statements, then the reified literal approach is correspondingly more compact for commonly referenced literals. The opposite is true if datatype properties are more compact in memory.

Our query speed comparison shows one type of reified literal queries that are faster than or equal to non-reified queries. This is due to the fact that there is a faster than or equal relationship between *(i)* an equijoin and *(ii)* a join equating two unreified literals. The other type of reified literal query is slowed by the speed associated with addition of a single equijoin. Further, we state that *(i)* an equijoin may be much faster than *(iii)* a join inexactly matching two unreified literals[2].

```
i:   {?person1 givenName ?reified_name .
      ?person2 givenName ?reified_name . }
ii:  {?person1 givenName ?unreified_name .
      ?person2 givenName ?unreified_name . }
iii: {?person1 givenName ?unreified_name1 .
      ?person2 givenName ?unreified_name2 .
      FILTER (likeTerm(?unreified_name1,
                       ?unreified_name2,
                       partialMatchSpec) }
```

Since the equijoin uses fast instance or integer comparison, it is faster or comparable to the join of two unreified literals[3].

[2] The *likeTerm* function is a non-standard extension to SPARQL for performing partial matching of strings. It is a more powerful version of the SQL *LIKE* keyword. The third argument is a partial match specification string that specifies the type of partial match and an optional matching template. In our implementation and others, inexact match is onerously slow compared to the speed of an equijoin. But the usage of a full text index, could make inexact search significantly faster.

[3] In RDF store implementations where strings are shared resources, the reified and unreified approaches are comparable because both are based on the matching of integers rather than strings.

The comparison of structural complexity we can base in part on statement counts and amortization of memory footprint comparison. But we ultimately rely on our reader's judgment as to the relative complexity.

The comparison of these sometimes negative costs against the benefits of increased expressivity is similarly subjective.

4 Case Study: Reified vs. Non-reified Names

First we treat the question of memory waste for reified names. Creating a new reified name each time a data source mentions that name would clearly waste memory. We, however, create a particular reified name instance just once for all its references to share. The cost of representing a non-reified name is one statement (<JohnDoe givenName "John">). The cost of a reified name would count the following object property statement (<JohnDoe givenName Name_JOHN>) plus the amortized cost of the shared reified name components for the name *John*: <Name_JOHN rdf:type Name> and <Name_JOHN value "John">. If just one reference to a person with the name "John" is in the data store, the extra reification cost is two statements – three times the non-reified approach. If the cost is shared between two references, the amortized extra cost is one statement; and among ten, the extra cost drops to one fifth of a statement (see **Figure 2**).

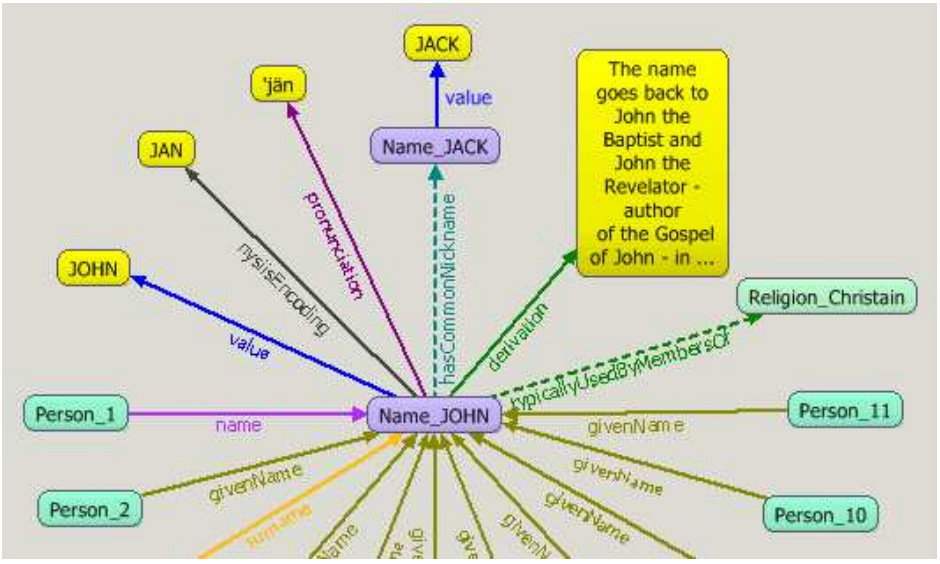

Figure 2: Because reified names are shared, the memory cost of the name *rdf:type* statement and name *value* statement is amortized out over all eleven name references. Note the several examples of the expressive power of reified names.

If names are common, the cost for their shared statements is negligible. So uncommon or *rare* reified names are individually costlier than unreified names, but the overall reified name cost is a function of the average number of references to the names.

Second, we treat the question of query speed. Exact name matching queries need not be string based. With name reification, one can match name instances rather than name strings. Implementations, then, can use integer comparison for speed equal to or faster than matching un-reified names.

When attempting an exact match query on a particular name, on the other hand, one must create the URI for that particular name, and one must create it in some repeatable canonical fashion. One must, for example, always translate the name *John* to precisely the same URI (e.g.: http://foo.gov/bar#Name_JOHN). This same name URI creation algorithm must be used for all names in the knowledge base. With these precautions, particular name matching also can also be a matter of simply comparing instances/integers rather than using an extra join to compare strings.

Inexact name matching requires an extra join when using reified names because the actual name string must be consulted. But this increase in query time is never large and is moot when using a system whose time for inexact matching overwhelms the cost of that extra join.

Next, we consider the structural complexity associated with reified names. The path length from a person node to her actual name value is one statement longer with the reified approach, but paying this price allows us to say things about names (see next section) in an organized fashion. We claim that if name meta-information is required, it is more intuitive to link it to the reified name instance and that the net effect is a reduction in complexity.

We now consider the benefits of reified names. Perhaps the most obvious benefit is being able to conveniently and intuitively say things about names and to reason about that information. A reified name can be linked directly to various information of interest to the IC such as its New York State Identification and Intelligence System (NYSIS) encoding, its variants, its nicknames, its ethnic derivation, a notion of its level of formality, its gender association, etc. (See **Figure 2**). Such information can be encoded in a system using non-reified literals, but the querier would need to understand the association between the name meta-information and the respective names. Clearly it is more intuitive to directly link the information about a name to some shared representation of the name. A newcomer need not know where the NYSIS information is stored. She simply queries on the name and sees, by inspection, that NYSIS information is associated with its corresponding names.

5 Generalizing Name Reification Results for Additional Literals

We discuss in detail the merits of reifying other literal types. We begin with dates, heights and weights.

As with reified names, reified dates are shared among all the events that reference them and, consequently, experience the same potential for memory cost amortization. Exact date match query speed, as for all reified literals, is at least comparable to the

non-reified case. Simple time range queries bounding the reified date's *xsd:date* value with two *xsd:date* values require an additional join Structural complexity of reified dates is clearly comparable to that of reified names. Expressivity-wise, dates are, of course, actual time intervals rather than simple time points. As an immediate result of date reification, one can concretely begin to better support temporal reasoning for IC applications. One can start by attaching beginning and end date-times to a date so as to allow dates to participate in precise time-interval-based queries. Reified dates can be unknown and yet known to be before some other known date. Such unknown date instances can participate readily in temporal interval overlap relations such as *temporallyContains* in order to bound the date if possible (see **Figure 3**).

With height and weight reification, amortized sharing cost efficiency is somewhat moot as likely usage tends toward few height and weight values[4]. Because even a minute difference in an exact match query is a miss, range queries are much more useful with scalars. Query speed in this case, therefore, is reduced by the cost of an equijoin, as actual weight and height values must be accessed. Reification increases expressivity such as being able to encode that one person is taller than another and both heights were unknown.

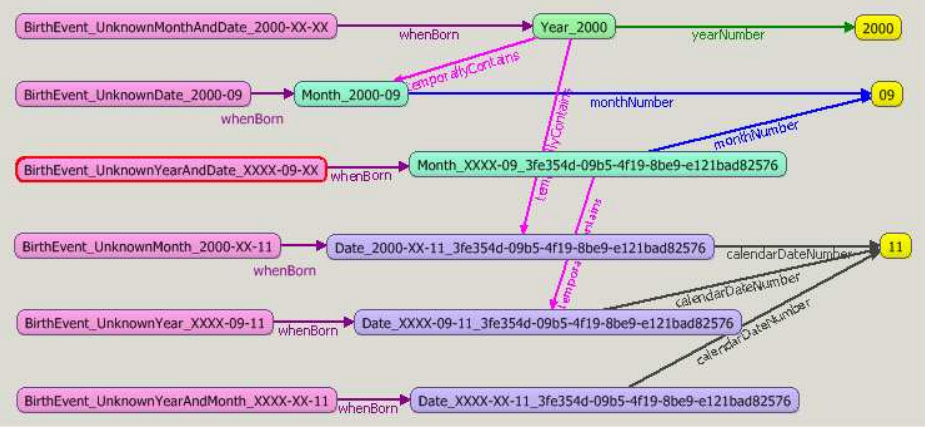

Figure 3: The following diagram shows six birth events each with a different sort of partial date. Years, months, dates, and birth events all have temporal extent and can, therefore, participate the various temporal interval algebra relations. Unknown date and months have URI names with embedded SHA 256 hash values to prevent them from coalescing with similar unknown dates and months. Were birthdays modeled as a datatype property rather than as reified dates, this sort of query-time expressivity would difficult and less intuitive.

These analyses apply to all twenty five of our twenty five literals. Without going into prohibited detail, all our literals are inherently sharable and, therefore, offer the same memory cost amortization potential. Zipf-Mandlebrot power law results are infrequently available for reifiable literals and they offer no guarantees that their exponents will be such that we can know that memory cost will be negligible[8]. All

[4] There are 289 English half inch values between zero and twelve feet inclusive.

but four of our date types and four of our scalars inherently lend themselves to using exact match and partial match, as with reified names. The reified name analysis, then, directly applies to these 17 literals. The two other scalar types behave just as height and weight with the range queries that are slower by one equijoin. The other date types are *Month, Year* and *TimePoint.* They are all simply date-like time intervals of various sizes, so their analyses are comparable to *Date*[5]

6 Results, Discussion, and Future Work

We established that common reified names have comparable in-memory cost to non-reified names. We similarly established that query speed for exact matching of reified names is equal or better than non reified names. And the additional speed cost of inexact matching is negligible in systems where inexact matching speed dominates that of a join. We argue that the overall structure of reified names and their metadata is simpler. We showed that reified names allow a sort of tightly linked expressivity that un-reified names do not.

We similarly analyzed dates, heights, and weights and found them to be slower by one join in interval queries. We found date expressivity to be significant and height and weight expressivity similar yet less likely to be justified by our data sets.

We distilled the following rules from the above analysis to help determine when the literal reification design pattern should be used:

1. Rare reified literals are individually costly, but the net cost is only a concern if there are very many rare types.
2. Range queries such as with reified scalars and dates are slower by an equijoin.
3. Inexact match queries over reified literals are slower by an equijoin. That equijoin is inconsequential on systems where inexact match dominates the query time.
4. Otherwise the speed and memory cost is comparable.

As we value expressivity, we found most of our literals to be reasonably strong candidates for reification. Our desire for expressivity also makes us less concerned as to how well amortized our shared structure is. We found all of our scalars to be weaker/marginal candidates because we have no present or near future need for scalar-related expressivity. Their inclusion would be based more on a desire to apply all design patters consistently.

In all cases, the value of the expressivity gain must be subjectively weighed against the possible cost in memory and speed. We have used literal reification for over fifteen years in the IC and in two different data integration projects *at scale*.

We expect that as we continue to observe the results of our choices to reify and not to reify literals, we will more finely characterize how to make such choices in the future. We expect to have opportunity to garner shared structure amortization statistics on our various reified literals.

[5] *TimePoint* is encoded as an interval as per common convention. *TimePoint* duration varies with the number of significant digits in the input.

7 Conclusions

Commonly referenced reified literals come at little or no significant cost in memory, speed, or complexity. Queries over such literals are never slower than the cost of one join with respect to unreified literals and are usually comparable. Where literal-related expressivity is specifically needed or expected, reified literals should be considered.

References

1. Cycorp Inc.: OpenCyc. http://opencyc.org
2. Hightfleet (formerly OntologyWorks): IODE, http://www.highfleet.com
3. Pease, A., Niles, I., and Li, J.: The suggested upper merged ontology: A large ontology for the semantic Web and its applications. In Proceedings of the AAAI-2002 Workshop on Ontologies and the Semantic Web, Edmonton, Alta., Canada (2002)
4. Masolo, C., Borgo, S., Gangemi, A., Guarino, N., Oltramari, A., and Schneider, L.: DOLCE: A Descriptive Ontology for Linguistic and Cognitive Engineering. WonderWeb Project, Deliverable D17 v2.1 (2003)
5. Hobbs, J., Pan, F.: Time Ontology in OWL. Working draft, http://www.w3.org/TR/owl-time (2006)
6. Charlet, J., Vandenbussche, P.: Concept Terms. Ontology Design Patterns. http://ontologydesignpatterns.org/wiki/Submissions:ConceptTerms
7. Gangemi, A.: Ontology Design Patters for Semantic Web Content. ISWC 2005. LNCS, vol. 1729, pp. 262-276 (2005)
8. Zipf, G., Selected Studies of the Principle of Relative Frequency in Language. Harvard University Press, Cambridge, MA (1932)