# Multiple Inheritance of Ontology Concepts in a Semantic-Aware Encoding Scheme

Weiqin Xu[1,2], Olivier Curé[1], Philippe Calvez[2]

[1] LIGM (UMR 8049), CNRS, UPEM, F-77454, Marne-la-Vallée, France.{firstname.lastname}@u-pem.fr
[2] ENGIE CRIGEN CSAI Lab, philippe.calvez1@engie.com

## 1 Introduction

Litemat[1] is semantic-aware encoding scheme that was originally designed to support efficient inferences over the RDFS ontology language. Intuitively, it encodes, with integer values, the elements of the TBox, i.e., concepts and properties, in such a way that reasoning over these hierarchies can be reduced to some computation over identifier intervals. It has recently been extended to support `owl:sameAs`[4], `owl:inverseOf` and `owl:transitiveProperty`[2] by proposing a similar form of encoding for elements of the ABox, i.e., individuals. We can hence consider that Litemat now supports inferences in the ontology language frequently referred to as RDFS++.

The main advantages provided by this encoding scheme is to considerably reduce the need for inference materialization and to provide an adapted query rewriting and optimization approach that improve the performance of query answering. Considering materialization, Litemat eliminates the need to store inferred facts since the semantics of attributed identifiers to concepts, properties and individuals are sufficient for RDFS++ reasoning services. Moreover, Litemat's query rewriting approach replaces costly union of SPARQL basic graph patterns (BGP) with query filters over integer intervals. Litemat hence proposes a trade-off between the two most popular inference solutions encountered in RDF stores. These nice properties only come with an overhead of maintaining slightly larger dictionaries, due to larger identifier values and additional metadata, than the standard case of most RDF stores.

In this paper, we present Litemat's support for multiple inheritance as well as the query processing and optimization that comes with it. [3]

## 2 LiteMat's encoding approach and query processing

In this section, we present LiteMat's encoding scheme using the following ontology concept hierarchy (but this method can also be applied to property hierarchies): $A \sqsubseteq \top$, $B \sqsubseteq \top$, $C \sqsubseteq \top$, $D \sqsubseteq \top$, $A1 \sqsubseteq A$, $C1 \sqsubseteq C$, $F1 \sqsubseteq F$, $E1 \sqsubseteq E$, $E2 \sqsubseteq E$, $F \sqsubseteq B$, $F \sqsubseteq C$, $E \sqsubseteq A1$, $E \sqsubseteq B$ and $E \sqsubseteq C$.

We can see that the last 5 concept subsumptions describe some multiple inheritance, i.e., the concepts $E$ (resp. $F$) have 3 (resp. 2) super classes.

In Figure 1(a), we present LiteMat's encoding for this TBox. In a first step, the assignment of an identifier, using a binary representation, for each concept is performed in a top-down recursive manner, i.e., it starts by setting the top concept ($\top$) at 1, and proceeds level-wise on the element hierarchy until all leaves have been processed. Intuitively, for each concept $\alpha$, we count the number N of direct sub concepts (including $\alpha$), e.g., in our running example $\top$ has 5 direct sub concepts. At this level, $\lceil \log(N) \rceil$ provides the number of bits necessary to represent each sub concept. Then, these sub concepts (excluding $\alpha$) are prefixed by the binary identifier of $\alpha$ and uniquely get a binary representation of a value $\in [1, N-1]$. For instance, the concept $A$ is prefixed with '1' ($\top$'s identifier and is assigned the value 1 on 3 bits, i.e., '001', yielding the binary string '1001').

Finally, a normalization step makes sure that all identifiers are encoded on the same binary string length. This is performed as follows: once all concepts have been encoded in the first step, we get the size L of the longest encoding string (i.e., 8 in our running example). Then all concept identifiers with an encoding length lower than L are appended with '0' until their length reaches L. This normalization step is represented with red '0' in Figure 1(a). The last column of this figure provides the integer identifier corresponding to each concept.

With this encoding scheme, it is clear that multiple inheritance poses a problem since each ontology concept must have a single identifier. For instance, in our running example, we can provide three different identifiers to $E$: one computed from $A1$, another one from $B$ and a last one from $C$. In the next section, we propose a solution to this issue.

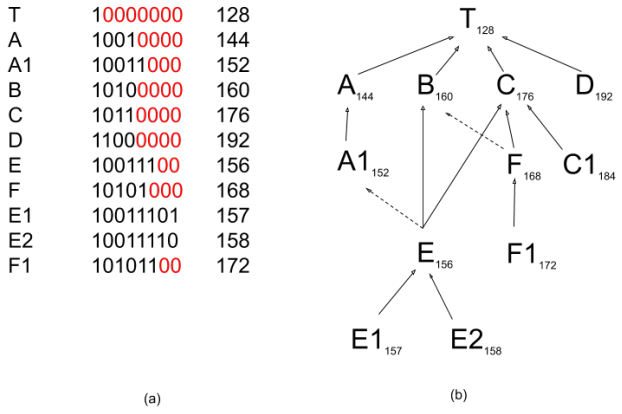


**Fig. 1.** LiteMat Encoding

LiteMat proposes an efficient query processing approach that takes advantage of the semantic-aware encoding. In fact, whenever a query requires to reason over the concept or property hierarchies, the system simply introduces new variables that are filtered on the identifiers of our encoding scheme.

Consider the following BGP of a SPARQL query: {?x type E}. It is generally rewritten into {?x type E} UNION {?x type E1} UNION {?x type E2}. In LiteMat, the system would identify that $E$ has several sub concepts, replace $E$ with a new variable and add a FILTER clause that restricts the accepted values of this new variable. This restriction corresponds to an interval of integer values where the lower bound is the identifier of $E$ and the upper bound is easily computing (i.e., using 2 bit shift operations and an addition) from the identifier of $E$. This computation requires an identifier metadata stating the index on the bit string where the normalization has started. Considering that the identifiers of $E$, $E1$ and $E2$ are resp. 156, 157 and 158 (to be explained in the next section), our LiteMat rewriting would be: {?x type ?y. FILTER (?y≥156 && ?y<157)}. This rewriting also applies when sub property relationships are used and the more complex the query, the more efficient our rewriting.

## 3 Multiple inheritance support in LiteMat

### 3.1 Encoding scheme

Our support of multiple inheritance is based on the notion of a representative. A representative $C_r$ is selected among the super concepts $C_1$,..,$C_n$ of a concept $SC$. That is the integer identifier of $SC$ will be computed following Litemat's approach based on the $C_r$ identifier. It is obvious that with this approach $SC$ loses any connection to its non-representative direct and indirect super concepts. Hence it is necessary to keep track of these super concepts. In the following, the remaining super classes of $SC$, i.e., $\{C_1,..,C_n\} \setminus C_r$ are considered as non-representative of $SC$.

Let consider that the representative of the concept $E$ is $A1$. Hence, the non-representatives of $E$ are the concepts $B$ and $C$. In Figure 1(b)(where each concept has its identifier in subscript), a representative is pointed by a dashed arrow and we can observe that the identifiers of concepts $E$, $E1$ and $E2$ (resp. 156, 157 and 158) are computed from $A1$'s identifier (i.e., 152).

To support an efficient query processing, we require a key/value data structure, denoted nonRep. Intuitively, each non representative of the ontology is an entry in that data structure and the value associated to a key corresponds to a set containing all the sub concepts involved in a multiple inheritance with the key concept as one of its super concept. In our running example, nonRep(B) = {E} and nonRep(C) = {E,F}.

### 3.2 Query processing and optimization

The query processing presented in Section 2 is extended to produce complete and correct result sets for ontologies involving multiple inheritance. The extension

coincides to the addition of disjunction in the generated FILTER clause of the rewritten SPARQL queries. Like in the original rewriting approach of Section 2, the disjuncts correspond to interval descriptions for a given query variable. An interval is computed using the nonRep data structure. Note that queries involving representatives do not necessarily involve this form of query processing. Due to space limitation, we present this processing and a simple optimization through the following example.

We consider the following BGP of a simple query, denoted $Q$, which involves a multiple inheritance: {?x rdf:type C}. The query rewriting denoted $Q'$ corresponds to: { ?x rdf:type ?y. filter((?y≥176 && ?y<192) || (?y≥156 && ?y<160) || (?y≥168 && ?y<176)) where the first disjunct corresponds to the standard interval defined in Section 2 and last two are computed using the nonRep data structures. That is, we search whether $C$ is involved in a multiple inheritance by checking its presence as a key in nonRep. If it is the case, it will return a set of concepts and for each of these concepts a disjunct is added to the FILTER clause over that variable. In our running example, nonRep($C$) returns a set with concepts $E$ and $F$, resp. the identifiers 156 and 168. These values correspond to the lower bounds of the intervals and upper bounds are computer as stated in LiteMat.

Based on the intervals present in this FILTER clause, a simple optimization can be performed. It has the possible effect of reducing the number of disjuncts in a query rewriting. The optimized queries are $Q''$ : {?x 0 ?y. filter((?y≥168 && ?y<192) || (?y≥156 && ?y<160)), i.e., it contains one less disjunct.

## 4 Conclusion

An encoding scheme, based on the semantic-aware approach of LiteMat, has been extended to support multiple inheritance. We emphasized that it can be used to efficiently to answer SPARQL queries. The overhead of the encoding scheme is relatively low and as been used in use case involving static and streaming data[3].

## References


1. Olivier Curé, Hubert Naacke, Tendry Randriamalala, and Bernd Amann. LiteMat: A scalable, cost-efficient inference encoding scheme for large RDF graphs. In *2015 IEEE International Conference on Big Data, Big Data 2015*, pages 1823–1830, 2015.
2. Olivier Curé, Weiqin Xu, Hubert Naacke, and Philippe Calvez. Extending LiteMat toward RDFS++. In *Extended Semantic Web Conference, LASCAR workshop*, 2019.
3. Jérémy Lhez, Xiangnan Ren, Badre Belabbess, and Olivier Curé. A compressed, inference-enabled encoding scheme for RDF stream processing. In *The Semantic Web - 14th International Conference, ESWC 2017*, pages 79–93, 2017.
4. Xiangnan Ren, Olivier Curé, Hubert Naacke, Jérémy Lhez, and Li Ke. Strider$^{\mathrm{r}}$: Massive and distributed RDF graph stream reasoning. In *2017 IEEE International Conference on Big Data, BigData 2017*, pages 3358–3367, 2017.