Optimizing Semantically Lifted Programs through Ontology Modularity

Eduard Kamburjan and Jieying Chen

SIRIUS Centre, Department of Informatics, University of Oslo, Oslo, Norway
{eduard,jieyingc}@ifi.uio.no

We investigate optimization of programs using Semantic Web techniques. A program is optimized by reducing the number of logical axioms needed to reason about it, based on the queries to be executed. Irrelevant axioms allow us to safely remove certain parts of the program.

Motivation. Semantic Web technologies [2] are a set of techniques to formally attach semantic meaning to data. They are used as a crucial step to integrate domain knowledge into many modern innovation applications such as AI expert systems and digital twins [5,9]. Among their core elements are description logic-based formalisms to model domain knowledge and store data using ontologies, as well as retrieve data from ontologies using special query languages.

Despite their numerous applications for static data, Semantic Web technologies are not suited to model dynamic processes and have no established mechanism for data changes. Furthermore, their implementations give little static guarantees to the programmer. To address this problem, Kamburjan et al. [3] introduced *semantically lifted programs*, which attach semantic meaning to a program state and, consequently, can access the program state using semantic web query languages by mapping a program state to an ontology and enriching it with domain knowledge. This tight integration allows, for example, to give static type guarantees to query results [4].

In this work, we investigate the use of ontology modularity[1] [6,8] to perform a sound program transformation on semantically lifted programs to increase their performance.

```
 1 class Task(...)...end
 2 class Server(List<Task> tasks)...end
 3 class Platform(List<Server> servers)...end
 4 class Scheduler(List<Platform> platforms)
 5  Unit reschedule()
 6   List<Platform> l:=
 7    access("SELECT ?x WHERE {?x a Busy}");
 8   this.optimizePlatforms(l);
 9  end
10 end
```

Semantically Lifted Programs. We introduce our approach using a running example in SMOL, an implementation of semantically lifted programs in a minimal object-oriented language. The figure on the right shows a (partial) example that models a cloud system scheduler: A scheduler monitors some set of platforms, each of which runs a set of servers, which in turn execute some tasks. The scheduler performs a SPARQL query in line 7 to retrieve all busy platforms. Note that the term `Busy` is *not* defined in the program itself. Instead, it is part of the domain knowledge that defines when a platform is considered busy. Thus, semantically lifted programs separate declarative domain knowledge and imperative computations using queries as interfaces.

To perform the query, we perform *semantic state access*: the current configuration of the program is translated into an ontology and enriched with the user-defined domain knowledge before executing the query. An ontology, in general, consists of a TBox and an ABox. A TBox is a finite set of general concept inclusions and role inclusions, while an ABox is a finite set of unary and binary assertions on individuals. The ABox must be *consistent* with the TBox, i.e., the assertion axioms of the ABox must be logically consistent with the inclusion axioms of the TBox. The following example shows a *part* of an ontology for our running example. The TBox

[1]In this work we stick to the terminology common in ontology modulularity and (1) do not distinguish between ontologies and knowledge graphs/bases, and (b) use the term modules *only* for ontologies, *not* programs.

$\mathcal{T}^\star$ has three axioms that describe (1) that concept `Busy` is a `Platform` and a `NonEmpty` (2) that a `NonEmpty` is an object which has a `servers` that is not empty, and (3) that a `Task` is a subclass of `Object`. The first two axioms represent domain knowledge, while the last is derived from the class table. The ABox $\mathcal{A}^\star$ defines a task, a platform, a list and connects them.

$$\begin{aligned}\mathcal{T}^\star = \{&\mathtt{Busy} \sqsubseteq \mathtt{Platform} \sqcap \mathtt{NonEmpty},\\ &\mathtt{NonEmpty} \sqsubseteq \exists\mathtt{servers}.\mathtt{NotNull}, \mathtt{Task} \sqsubseteq \mathtt{Object}\}\\ \mathcal{A}^\star = \{&\mathtt{Platform}(\mathsf{a}), \mathtt{List}(\mathsf{b}), \mathtt{Task}(\mathsf{c}), \mathtt{servers}(\mathsf{a}, \mathsf{b})\}.\end{aligned}$$

To generate the ontology, the current program state conf is semantically lifted by a function μ to an *ABox* $\mathcal{A} = \mu(\mathsf{conf})$. The TBox $\mathcal{T} = \mathcal{T}_{\mathtt{SMOL}} \cup \mathcal{T}_{\mathsf{user}}$ of the used ontology is static throughout program execution and contains axioms $\mathcal{T}_{\mathtt{SMOL}}$ describing the class table and the domain ontology $\mathcal{T}_{\mathsf{user}}$. Finally, the query q is evaluated using some answering engine ans to retrieve a list of elements from the knowledge graph. The answering engine must perform reasoning steps to derive new data using the TBox axioms and commonly includes a description logic reasoner. The runtime semantics of **access** in a configuration conf computes the following:

$$\mathsf{ans}\big((\mathcal{T}, \mathcal{A}),\ q\big) = \mathsf{ans}\big((\mathcal{T}_{\mathtt{SMOL}} \cup \mathcal{T}_{\mathsf{user}}, \mu(\mathsf{conf})),\ q\big).$$

Using Modules for Sound Program Transformation. Semantic state access is a main performance bottleneck, as each such access requires performing reasoning steps in the query engine. Thus, it is of major importance to optimize programs with respect to semantic state access. The approach we present here is *localization*: instead of using the same direct mapping μ and TBox $\mathcal{T}$ everywhere, we apply a program transformation such that each **access** statement has a local direct mapping μ' and TBox $\mathcal{T}'$. This increases the performance even if a program contains only one **access** statement, as it reduces the size of the knowledge graph by adapting it to a specific query. For localization, we use the theory of ontology modularity.

In general, a module of an ontology wrt. the given vocabulary is a sub-ontology that still preserves certain logical consequences about the given vocabulary. We call this vocabulary a signature, denoted as Σ, which is a set of concept and property names. Here, we only extract the modules on TBoxes. Formally, we say $\mathcal{M} \subseteq \mathcal{T}$ is a module of $\mathcal{T}$ wrt. Σ if the following statements hold for all queries q, $sig(q) \subseteq \Sigma$ and any ABoxes $\mathcal{A}$ that are consistent with (the self-consistent) $\mathcal{T}$: $\mathsf{ans}\big((\mathcal{T}, \mathcal{A}), q\big) = \mathsf{ans}\big((\mathcal{M}, \mathcal{A}), q\big)$, where $sig(q)$ is the signature of q. Specially, there are different module notions that specify what kind of (logical) queries that modules should preserve, such as semantic modules [6] and deductive modules [1,7]. However, it is usually very difficult to compute such modules. Locality-based modules [8] was proposed as a practical solution to compute ontology modules. Since they preserve almost all logical consequences and it takes polynomial time to exact such modules. Considering the example that we introduced above, we have $\Sigma^\star = \{\mathtt{Busy}\}$. Then the set of first two axioms of $\mathcal{T}^\star$ is the module (under any module notion) of $\mathcal{T}^\star$ wrt. $\Sigma^\star$.

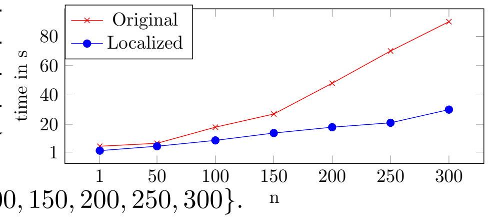
We can intuitively see that no axioms describing the `Task` and `Scheduler` classes are needed. Going beyond the TBox, we can use the module mechanism to also compute precisely which axioms from the ABox can be removed: After computing $\mathcal{M}$, the module of (T) wrt. $\Sigma(q)$ we can also modify μ and suppress the generation of any ABox axioms that does not include symbols from $\Sigma(\mathcal{M})$, e.g., not generating the axiom $\mathtt{Task}(c)$.

The soundness of the localization is ensured. That is, computing the original query q on the computed module and any ABox returns the same results, for *every* possible ABox $\mathcal{A}$ (that is consistent with $\mathcal{T}$), even when restricted to the vocabulary of $\mathcal{M}$, (written $\mathcal{A}\restriction\Sigma(\mathcal{M})$):

$$\mathsf{ans}\big((\mathcal{T}, \mathcal{A}),\ q\big) = \mathsf{ans}\big((\mathcal{M}, \mathcal{A}\restriction\Sigma(\mathcal{M})),\ q\big) \qquad \text{where } \mathcal{M} \text{ is the module of } \mathcal{T} \text{ wrt. } \Sigma(q).$$

Let P' be the localized variant of P, a copy of P where each **access**(q) statement is replaced by a localized **access**(q,$\mathcal{M}$,$\mu_{\mathcal{M}}$) with the obvious runtime semantics. From the definition of modules, it follows that $P \equiv P'$, i.e., that both programs reach the same final states. Localization is static: neither program nor query is executed. Furthermore, it goes beyond preprocessing of queries at the query engine, as we perform the optimization (1) *before* the axioms are generated instead of deleting them and (2) *per query* and not *per query execution*.

Ontology modules can be used to improve performance by reducing the number of considered axioms. As we see in the following, localization allows to use the performance benefits for semantically lifted programs. We perform a preliminary evaluation using the 2-3 tree benchmark from [3] with n elements added to a 2-3 tree, for $n \in \{1, 50, 100, 150, 200, 250, 300\}$. Afterwards, a SPARQL query with the Apache Jena framework retrieves all 2-nodes, which are expressed in an OWL ontology and logical reasoning is required to retrieve them. As the graph above shows, already a coarse modularization can decrease runtime by 61%.



We remind that the axioms in $\mathcal{T}_{\mathsf{SMOL}}$ are modeling programs elements. Consider the situation where the aim of the program is to examine its final state using queries – if certain axioms are not part of any module, including the module needed for the final queries, then it is safe to not compute them. For example, if a field does not occur in any modules, then it is possible to not only adjust the lifting function, but also to remove all read and write accesses to this field that do not effect the final state (of other fields). On-going work suggest that such optimization can be performed by integrating ontology modules with adjusted program slicing techniques.

Outlook. This work presents *localization*, a sound program transformation of semantically lifted programs, and preliminary empirical results that suggest that it is increasing performance. Beyond on-going work on program slicing, we plan to investigate which kind of ontology modules are most suited for performance optimization in the setting of semantically lifted programs.

Acknowledgments. This work is funded by `SIRIUS` (RCN 237898) and `PeTWIN` (RCN 294600). We thank Martin Giese and Einar Broch Johnsen for comments and feedback on this work.

References

[1] J. Chen, M. Ludwig, Y. Ma, and D. Walther. Zooming in on ontologies: Minimal modules and best excerpts. In *ISWC*, volume 10587 of *LNCS*, 2017.

[2] P. Hitzler et al. *Foundations of Semantic Web Technologies*. Chapman and Hall/CRC Press, 2010.

[3] E. Kamburjan, V. N. Klungre, R. Schlatte, E. B. Johnsen, and M. Giese. Programming and debugging with semantically lifted states. In *ESWC*, volume 12731 of *LNCS*, 2021.

[4] E. Kamburjan and E. V. Kostylev. Type checking semantically lifted programs via query containment under entailment regimes. In *Description Logics*, volume 2954 of *CEUR*, 2021.

[5] E. Kharlamov et al. Towards semantically enhanced digital twins. In *IEEE BigData*. IEEE, 2018.

[6] B. Konev, C. Lutz, D. Walther, and F. Wolter. Model-theoretic inseparability and modularity of description logic ontologies. *Artificial Intelligence*, 203:66–103, 2013.

[7] P. Koopmann and J. Chen. Deductive module extraction for expressive description logics. In *IJCAI*. ijcai.org, 2020.

[8] U. Sattler, T. Schneider, and M. Zakharyaschev. Which kind of module should I extract? In *Description Logics*, volume 477 of *CEUR*, 2009.

[9] B. Zhou et al. Predicting quality of automated welding with machine learning and semantics: A bosch case study. In *CIKM*. ACM, 2020.