Incremental and Persistent Reasoning in FaCT++

Dmitry Tsarkov

tsarkov@cs.man.ac.uk
School of Computer Science
The University of Manchester
Manchester, UK

Abstract. Reasoning in complex DLs is well known to be expensive. However, in numerous application scenarios, the ontology in use is either never modified at all (e.g., in query answering), or the amount of updates is negligible in comparison with the whole ontology (e.g., minor manual edits, addition of a few individuals). In order to efficiently support these scenarios, FaCT++ implements the following two techniques: persistent and incremental reasoning. In *persistent reasoning* mode, after classification, the reasoner saves its internal state, including computed information (e.g., concept taxonomy) on a persistent medium; the next use of the ontology will not require classification to be performed from scratch, but just reading an input file. In *incremental reasoning* mode, the reasoner is notified of a change and identifies a (usually small) portion of its internal state that is affected by the change. This is the only part that requires recomputation. This approach can lead to greater overall efficiency, when compared with having to reload and reclassify the whole ontology.

1 Introduction

Reasoning about ontologies in the OWL 2 DL profile is computationally very expensive. The underpinning logic has 2NEXPTIME worst case complexity for common reasoning tasks, which makes reasoning for these ontologies highly intractable. However, while this complexity cannot be optimized away, there are various scenarios in which a reasoner can avoid doing the same work multiple times.

One such scenario assumes that the ontology does not change at all or changes rarely and in a predictable manner; this is the case for ontologies that stay unchanged between releases, and are released every few weeks, rather than being edited multiple times per day. Examples of such a scenario could be ontology-based applications that ask hierarchical queries, without additions or removals of axioms and assertions. In this case there is no need to re-classify the ontology each time the system runs, as the result of classification would stay the same for long periods of time.

Another common scenario assumes that the ontology changes frequently but the changes are much smaller than the ontology itself. A typical example of

such a scenario is manual ontology editing. Full reclassification in this context is suboptimal, as only a minor part of the concept hierarchy will be changing between reclassifications.

To support these scenarios we implement additional reasoning modes in an OWL 2 DL reasoner FaCT++ [5]. In *persistent mode*, FaCT++ saves the inferred information together with its internal state into a file, which can then be reloaded with much less computational effort than reasoning would require. In *incremental mode*, FaCT++ carefully determines which parts of the precomputed inferences may be affected by an incoming change and only recomputes a subset of the inferences.

In this paper we describe in detail how persistent and incremental reasoning modes are implemented in FaCT++ along with an empirical evaluation, and outline future developments.

2 Persistent Reasoning

We call a reasoner *persistent* if it can save its internal state together with some precomputed inferences (e.g., concept hierarchy) and reload it when presented with the same ontology. The purpose of persistent reasoning is to avoid duplication of expensive calculations for an input that has been already processed. One can view a persistent reasoner as a cache that is indexed by ontology. For every ontology in cache it returns a reasoner that is ready to answer queries about that ontology.

A good use-case for persistent reasoning is a query answering over a complex ontology, which is subject to changes only very rarely. For example, the SNOMED-CT terminology[1] is used in many healthcare applications, and is updated with a new release twice a year. Although this ontology is in the OWL 2 EL profile, which means classification complexity is polynomial and it can be classified quickly with tractable EL reasoners, an OWL 2 DL ontology of comparable size can be problematic to deal with.

The implementation of a persistent reasoner depends a lot on the programming language and/or infrastructure around it. Some languages, like Java, allow one to (de-)serialize the internal state of the program as a language feature. However, FaCT++ is implemented in C++, so a different solution is needed. The well-known Boost library collection[2] contains a `Boost::Serialization` library that allows one to serialise/deserialise classes in a customisable way. In FaCT++ however we implemented our own version of serialising routines to have the necessary flexibility.

The most obvious approach to persistent reasoning is to save the entire internal state of the reasoner and reload it every time the same ontology needs to be classified or queried. This approach, however, seems to be too complicated and error-prone, as every minor change of the object model will lead to change in the (de-)serialising code. This also implies that the saved state cannot be loaded

[1] `http://www.ihtsdo.org/snomed-ct/`
[2] `http://www.boost.org/`

by two slightly different versions of the reasoner code, just as it happens with standard Java serialization.

The essence of the approach used in FaCT++ is to save and restore only inferred information, performing the initialisation of the internal state each time during the reasoner creation. This approach might look not optimal, however it provides a good compromise between minimising loading time and complexity of the implementation, combined with resilience to minor changes in the reasoner code. Typically, code using the reasoner must already have loaded the ontology in memory; it is therefore of little use to save the ontology data together with the reasoner state; not saving the ontology requires extra initialization work for the reasoner, but saves effort. This, as shown in Section 4, demonstrates good results in practice. This is due to the fact that preprocessing and consistency checking usually requires negligible time comparing to classification.

Let us have a look at the approach in a nutshell. After the classification or realisation is finished, FaCT++ saves the most important information of the internal state to a file. These include the signature of the ontology, the set of all expressions and the result of reasoning (e.g. the concept hierarchy, types of individuals, etc).

During the reload of the persisted state, the ontology is loaded as usual, and its consistency is checked. After this step, all internal structures have been initialised and the reasoner is ready to answer queries. This is the point at which the saved structures are loaded from the file. The reasoner then performs verification to ensure that the loaded ontology correspond to the saved one. This verification mainly compares the set of expressions: if the in-memory version coincides with the saved one (up to things like the ordering of elements in conjunctions), then the ontology is assumed to be the same.

Some preliminary tests of persistent reasoning can be found in Section 4.

3 Incremental Reasoning

The idea of an *incremental reasoning* mode is to avoid reloading and reclassifying the whole ontology when just a few axioms in the ontology are changed. The standard way of achieving this for tableaux-based reasoners is to use *modules* to determine, within an acceptable approximation, the affected subsumptions [1]. We use a similar approach in our implementation of incremental reasoning.

We now briefly introduce the notion of a *module* in DL. Let $\mathcal{O}$ be an ontology and Σ a *signature*, which is essentially a set of entities. The subset M of $\mathcal{O}$ is called a *module of* $\mathcal{O}$ *w.r.t.* Σ if for any axiom α with $Sig(\alpha) \subseteq \Sigma$, $M \models \alpha \iff \mathcal{O} \models \alpha$. The module M_α is a *module for an axiom* α if it is a module w.r.t. $Sig(\alpha)$. The precise definitions of module and the notion of $\bot$-locality-based module could be found, e.g., in [1, 4].

The approach presented in [1] provides an algorithm for incremental classification, i.e., it tries to minimise work that is necessary to provide a new concept hierarchy. The approach is based on the following propositions:

Proposition 1 (Proposition 1 from [1]). *Let $\mathcal{O}^1, \mathcal{O}^2$ be ontologies, α an axiom, and $\mathcal{O}^1_\alpha, \mathcal{O}^2_\alpha$ respectively modules for α in $\mathcal{O}^1$ and $\mathcal{O}^2$. Then, the following properties hold:*

- *If $\mathcal{O}^1 \models \alpha$ and $\mathcal{O}^1_\alpha \subseteq \mathcal{O}^2$, then $\mathcal{O}^2 \models \alpha$.*
- *If $\mathcal{O}^1 \not\models \alpha$ and $\mathcal{O}^2_\alpha \subseteq \mathcal{O}^1$, then $\mathcal{O}^2 \not\models \alpha$.*

Proposition 2 (Proposition 3 from [1]). *Let $\mathcal{O}$ be an ontology, α an axiom in the form $A \sqsubseteq B$, where $A, B \in Sig(\mathcal{O})$ are concept names. Then a module for axiom α can be computed as a $\bot$-locality-based module for signature $\{A\}$, denoted $\mathcal{O}_A$.*

The incremental classification algorithm works in several steps. First it takes the changes (the set of axioms that were added to $\mathcal{O}$ or removed from $\mathcal{O}$) and determines the set of concept names, for which the concept hierarchy can be altered due to that changes. Then for every such name A the algorithm rebuilds the module $\mathcal{O}_A$, taking into account ontology changes. If the module $\mathcal{O}_A$ does not change then, by Proposition 1, the concept hierarchy for A stays the same. If the module $\mathcal{O}_A$ changes, then the algorithm checks whether $\mathcal{O}_A \models \alpha$, where $\alpha = A \sqsubseteq B$.

In FaCT++ we use the algorithm described above with a few improvements.

The first one is about the module extraction approach. While locality-based module extraction is computationally cheap (the complexity is $O(n^2)$ for the algorithm from [1] and $O(n * m)$ for the one from [4], where $n = |\mathcal{O}|$ and $m = max(|Sig(\alpha)|)$ for $\alpha \in \mathcal{O}$), the number of modules in question can be large. Therefore, we use an approach inspired by the following observation (that is a consequence of Proposition 2 from [4]):

Proposition 3. *Let A and B be concept names such that $B \in Sig(\mathcal{O}_A)$. Then $\mathcal{O}_B \subseteq \mathcal{O}_A$. In other words, $\mathcal{O}_B = (\mathcal{O}_A)_B$.*

This leads to a stratified approach to module extraction. Once we find $\mathcal{O}_A$ for some A, we look at those $B \in Sig(\mathcal{O}_A)$ for which the module should be recalculated. Then we extract $\mathcal{O}_B$ as $(\mathcal{O}_A)_B$. We proceed recursively, so at all times we start to extract a module from as small a set of axioms as possible.

Another optimization we apply is the one described in Section 2. Namely, after having a change-set in hand, we save the internal state of the reasoner (which is a concept hierarchy in this case), reload the new ontology and perform preprocessing and a consistency check. After this the reasoner is ready to answer queries about the new ontology. The reasoner then loads the old hierarchy and performs changes according to the incremental algorithm described above. The fact that we use the whole $\mathcal{O}$ instead of $\mathcal{O}_A$ to query about subsumptions of the form $A \sqsubseteq B$ might look like a bad choice, taking into account the high complexity of reasoning. However, in this approach there is no need to organise separate reasoners for modules, load axioms in, etc. Moreover, the fact that the current instance of the reasoner is loaded with the whole ontology allows it to answer arbitrary queries about the changed ontology. This makes FaCT++

a complete incremental reasoner, unlike Pellet, which, while implementing the algorithm from [1], can only answer named concept subsumption queries; this makes it, in fact, an incremental classifier.

Additional benefit is the ability to determine non-subsumption quickly using modules. The Proposition 2 has the following very important corollary:

Proposition 4. *Let $\mathcal{O}$ be an ontology, A, B be concept names. Then if $B \notin Sig(\mathcal{O}_A)$, then $\mathcal{O}_A \not\models A \sqsubseteq B$.*

In other words, all the super-concepts of A are in the signature of $\mathcal{O}_A$. As long as we maintain $\mathcal{O}_A$ for all $A \in Sig(\mathcal{O})$, we can use that information to reduce the number of subsumption tests not only in an incremental step of reasoning, but also for the initial classification. The benefit of this could be huge, see Section 4 for the details.

4 Empirical Evaluation

All the experiments were done on the machine with Intel Core 2 Duo 3.06 GHz processor and 8 Gb of RAM under Max OS X 10.9.1. We used for the experiments FaCT++ [5] version 1.6.3 beta.

4.1 Persistent Reasoning

For the persistent reasoning tests we use a command-line version of FaCT++. For every ontology, we load it into a fresh reasoner, classify or realise it and then save the inferred information. On the second run the ontology is loaded and checked for consistency. After this, the inferred information is already loaded, so the reasoner is ready to answer hierarchical queries.

The results for a few of the most complex ontologies are presented in Table 1. The SNOMED CT ontology has a simple structure but very large size. The NCI Thesaurus ontology[3] is large and not so simple as SNOMED. The Family History ontology, FHKB [3] is very small but it uses unusually complex role hierarchy and is rather hard for modern reasoners. The columns in the table correspond to the time required to load the ontology into the reasoner, preprocess the ontology, check its consistency, classify or realise it, save it to a file and read the saved structures from that file (with verification). The ratio column shows how much faster the reasoning via restoring previously computed inferences is comparing to the actual classification. It is also easy to see that in the normal reasoning the classification time dominates over the initialisation of the reasoner. This is an empirical justification of the chosen approach.

4.2 Incremental Reasoning

The inspiration for our implementation of incremental reasoning comes from the joint project between Siemens Healthcare and the University of Manchester. In this project, reasoning was used in a complex ontology-driven application, to

[3] `http://ncit.nci.nih.gov/`

Ontology	$\|Sig(\mathcal{O})\|$	Initial reasoning time, sec				save/load time, sec		ratio
		load	preprocess	consistency	realise	save	load	
SNOMED	379,689	3.07	3.33	0.07	481.28	6.91	8.23	33
NCIt-10.05	98,902	0.79	1.35	0.12	568.49	2.17	2.68	115.5
FHKB	462	0	0	0.78	43.61	0	0	57

Table 1. Time of different reasoning and (de-)serialisation steps, in seconds.

answer queries that were written as conjunctions of complex concept expressions. The application asks for the super-concepts of these expressions and, depending on the answer, performs some actions.

The ontology developed in this project was classified using FaCT++, in non-incremental mode; this requires more than 8 hours. However, using incremental mode, the initial classification time is reduced to 20 minutes, most of which are spent in the module extraction procedure.

The reasons of such a disparity was found to be that the axioms that define the set of criteria look like $\alpha = C \sqsubseteq A$, where A is a criterion, and C is a complex expression, defining the criterion. Note that α is the only axiom that mention A. The ontology contains about 30,000 such axioms, most of which share some parts of their definitions. So, the reasoner in standard mode tries to check subsumption between different criteria, spending a lot of time there. However, $\mathcal{O}_A$ for such an A would be empty, and the incremental reasoner does not need to perform any classification over such concepts.

In order to test incremental reasoning we add/remove a small number (1-10) of criteria definitions at random. In all cases the reclassification time was less than 20 seconds, the mean was around 5 seconds, which shows a ratio of about 240 to 1 compared to the original classification time of 20 minutes.

5 Future Work

While currently implemented features of persistent and incremental reasoning work quite well, we can see future improvements for these techniques in several possible directions.

- It might be worth trying to use Labelled Atomic Decomposition (LAD) instead of modules for single concepts. The *LAD* is a compact representation of all modules of an ontology [7]. It is used in the Chainsaw reasoner [6] to extract modules. It might be beneficial to have a single representation of all modules rather than keep each one separately. However, it requires a fully incremental LAD implementation, while currently only incremental additions, but not removals, are allowed [2].
- Investigate the case of tighter integration of persistent and incremental reasoning. The obvious candidate for serialization is the set of modules. It might also be possible to save the history of changes, so the retraction of a change would just amount to the loading of a previous state.

– A more ambitious goal is to combine the two modes and use the result as a default mode for applications like ontology editors. An editor usually works with several ontologies, and changes tend to be minor. Such a reasoning system will save all the classified information, and load it using hashing over the ontology name and/or content.

6 Conclusions

We have identified two common ontology usage scenarios for which the performance of traditional reasoners is sub-optimal. We propose two reasoning modes, namely persistent reasoning and incremental reasoning, to improve reasoner performance in these scenarios. We described the implementation of these modes in the OWL 2 DL reasoner FaCT++ and evaluated our implementation on several real-life ontologies. Finally, a few directions for future work have been introduced.

References

1. Grau, B.C., Halaschek-Wiener, C., Kazakov, Y., Suntisrivaraporn, B.: Incremental classification of description logics ontologies. J. Autom. Reasoning 44(4), 337–369 (2010)
2. Klinov, P., Del Vescovo, C., Schneider, T.: Incrementally updateable and persistent decomposition of owl ontologies. In: OWLED (2012)
3. Stevens, R., Stevens, M.: A family history knowledge base using OWL 2. In: OWLED (2008)
4. Tsarkov, D.: Improved algorithms for module extraction and atomic decomposition. In: Proc. of 25th International Workshop on Description Logics (DL 2012) (2012)
5. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: Proc. of the International Joint Conference on Automated Reasoning (IJCAR 2006). Lecture Notes in Artificial Intelligence, vol. 4130, pp. 292–297. Springer (2006)
6. Tsarkov, D., Palmisano, I.: Chainsaw: a metareasoner for large ontologies. In: Proc. of OWL Reasoner Evaluation Workshop (ORE 2012). Manchester, UK (2012)
7. Vescovo, C.D., Parsia, B., Sattler, U., Schneider, T.: The modular structure of an ontology: Atomic decomposition and module count. In: WoMO. pp. 25–39 (2011)