ELK: A Reasoner for OWL EL Ontologies
(Technical Report)

Yevgeny Kazakov[1], Markus Krötzsch[2], and František Simančík[2]

[1] Institute of Artificial Intelligence, Ulm University, Germany
[2] Department of Computer Science, University of Oxford, UK

Abstract. ELK is a specialized reasoner for the lightweight ontology language OWL EL. The practical utility of ELK is in its combination of high performance and comprehensive support for language features. At its core, ELK employs a consequence-based reasoning engine that can take advantage of multi-core and multi-processor systems. A modular architecture allows ELK to be used as a stand-alone application, Protégé plug-in, or programming library (either with or without the OWL API). This system description presents the current state of ELK.

1 The System Overview

The logic-based ontology language OWL is becoming increasingly popular in application areas, such as Biology and Medicine, which require dealing with a large number of technical terms. For example, medical ontology SNOMED CT provides formal description of over 300,000 medical terms covering various topics such as diseases, anatomy, and clinical procedures. Terminological reasoning, such as automatic classification of terms according to subclass (a.k.a. 'is-a') relations, plays one of the central role in applications of biomedical ontologies. To effectively deal with large ontologies, several profiles of the W3C standard OWL 2 have been defined [7]. Among them, the OWL EL profile aims to provide tractable terminological reasoning. Specialized OWL EL reasoners, such as CEL [1], Snorocket [5], and jCEL [6], can offer a significant performance improvement over general-purpose OWL reasoners.

This paper describes the ELK system (http://elk-reasoner.googlecode.com/). ELK is developed to provide high performance reasoning support for OWL EL ontologies. The main focus of the system is (i) extensive coverage of the OWL EL features, (ii) high performance of reasoning, and (iii) easy extensibility and use. In these regards, ELK can already offer advantages over other OWL EL reasoning systems mentioned above. For example, as of today, ELK is the only system that can utilize multiple processors/cores to speed up the reasoning process, which makes it possible to classify SNOMED CT in as little as 5 seconds on a commodity hardware [2]. This paper does not provide any new theoretical results or experimental comparisons with other tools; those can be found in our earlier works [2–4].

ELK is a flexible system that can be used in a variety of configurations. This is supported by a modular program structure that is organized using the Apache

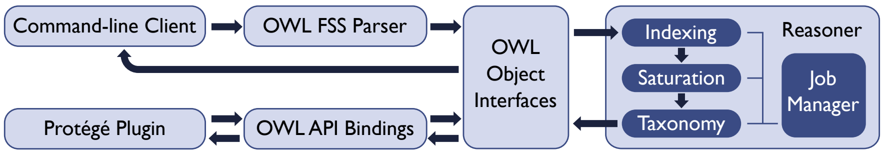

Fig. 1. Main software modules of ELK and information flow during classification

Maven build manager for Java. Maven can be used to automatically download, configure, and build ELK and its dependencies, but there are also pre-built packages for the most common configurations. The modular structure also separates the consequence-based reasoning engine from the remaining components, which facilitates extension of the system with new language features. The latest stable release ELK 0.2.0 supports conjunction (ObjectIntersectionOf), existential restriction (ObjectSomeValuesFrom), the top class (owl:Thing), complex role inclusions (property chains), and syntactic datatype matching. Support for nominals (oneOf), ABox facts (assertions), and datatypes is under development.

The main software modules of ELK are shown in Fig. 1. The arrows illustrate the information flow during classification. The two independent entry points are the command-line client and the Protégé plug-in to the left. The former extracts OWL ontologies from files in OWL Functional-Style Syntax (FSS), while the latter uses ELK's bindings to the OWL API[3] to get this data from Protégé.[4] All further processing is based on ELK's own representation of OWL objects (axioms and expressions) that does not depend on the (more heavyweight) OWL API. The core of ELK is its reasoning module, which will be discussed in detail.

Useful combinations of ELK's modules are distributed in three pre-built packages, each of which includes the ELK reasoner. The *standalone client* includes the command-line client and the FSS parser for reading OWL ontologies. The *Protégé plugin* allows ELK to be used as a reasoner in Protégé and compatible tools such as *Snow Owl*.[5] The *OWL API bindings* package allows ELK to be used as a software library that is controlled via the OWL API interfaces.

2 Preliminaries

The vocabulary of $\mathcal{EL}^+$ consists of countably infinite sets of *(atomic) roles* and of *atomic concepts*. Complex *concepts* and *axioms* are defined recursively using the constructors in Table 1. We use the letters R, S for roles, C, D, E for concepts and A, B for atomic concepts. A *concept equivalence* $C \equiv D$ abbreviates the two inclusions $C \sqsubseteq D$ and $D \sqsubseteq C$. An *ontology* is a finite set of axioms.

Given an ontology $\mathcal{O}$, we write $\sqsubseteq^*_{\mathcal{O}}$ for the smallest reflexive transitive binary relation over roles such that $R \sqsubseteq^*_{\mathcal{O}} S$ holds for all $R \sqsubseteq S \in \mathcal{O}$. We say that a

[3] http://owlapi.sourceforge.net/
[4] http://protege.stanford.edu/
[5] http://www.b2international.com/portal/snow-owl

Table 1. Syntax and semantics of $\mathcal{EL}^+$

	Syntax	Semantics
Roles:		
atomic role	R	$R^\mathcal{I}$
Concepts:		
atomic concept	A	$A^\mathcal{I}$
top	$\top$	$\Delta^\mathcal{I}$
conjunction	$C \sqcap D$	$C^\mathcal{I} \cap D^\mathcal{I}$
existential restriction	$\exists R.C$	$\{x \mid \exists y : \langle x, y\rangle \in R^\mathcal{I} \wedge y \in C^\mathcal{I}\}$
Axioms:		
concept inclusion	$C \sqsubseteq D$	$C^\mathcal{I} \subseteq D^\mathcal{I}$
role inclusion	$R \sqsubseteq S$	$R^\mathcal{I} \subseteq S^\mathcal{I}$
role composition	$R_1 \circ R_2 \sqsubseteq S$	$\langle x, y\rangle \in R_1^\mathcal{I} \wedge \langle y, z\rangle \in R_2^\mathcal{I} \rightarrow \langle x, z\rangle \in S^\mathcal{I}$

concept C *occurs negatively* (resp. *positively*) in an ontology $\mathcal{O}$, if C is a syntactic subexpression of D (resp. E) for some axiom $D \sqsubseteq E \in \mathcal{O}$.

$\mathcal{ELH}$ has Tarski-style semantics. An *interpretation* $\mathcal{I}$ consists of a nonempty set $\Delta^\mathcal{I}$ called the domain of $\mathcal{I}$ and an interpretation function $\cdot^\mathcal{I}$ that assigns to each R a binary relation $R^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$ and to each A a set $A^\mathcal{I} \subseteq \Delta^\mathcal{I}$. The interpretation function is extended to complex concepts as shown in Table 1.

An interpretation $\mathcal{I}$ *satisfies* an axiom α (written $\mathcal{I} \models \alpha$) if the corresponding condition in Table 1 holds. If an interpretation $\mathcal{I}$ satisfies all axioms in an ontology $\mathcal{O}$, then $\mathcal{I}$ is a *model* of $\mathcal{O}$ (written $\mathcal{I} \models \mathcal{O}$). An axiom α is a *consequence* of an ontology $\mathcal{O}$ (written $\mathcal{O} \models \alpha$) if every model of $\mathcal{O}$ satisfies α. A concept C is *subsumed* by D w.r.t. $\mathcal{O}$ if $\mathcal{O} \models C \sqsubseteq D$.

For atomic concepts A and B, the subsumption $\mathcal{O} \models A \sqsubseteq B$ is *direct* if there exists no atomic concept C nonequivalent to A and B w.r.t. to $\mathcal{O}$ such that $\mathcal{O} \models A \sqsubseteq C$ and $\mathcal{O} \models C \sqsubseteq B$. *Classification* is the task of computing the class *taxonomy* that represents the direct subsumptions between equivalence classes of atomic concepts occurring in $\mathcal{O}$.

3 Inference Rules

The ELK reasoning component works by deriving consequences of ontological axioms under inference rules. The improvement and extension of these rules is an important part of the ongoing development of ELK [2–4]. To simplify the presentation, in this paper we focus on inference rules for $\mathcal{EL}^+$, a small yet non-trivial fragment of OWL EL, which is sufficient to illustrate the work of the main reasoning component of the ELK system.

Inference rules for $\mathcal{EL}^+$ are shown in Fig. 2. The rules operate with expressions of the form $C \sqsubseteq D$, $C \overset{R}{\rightarrow} D$, and $\mathsf{init}(C)$. The axiom $C \overset{R}{\rightarrow} D$ has the same semantics as $C \sqsubseteq \exists R.D$, but is used differently in the inference rules. The expression $\mathsf{init}(C)$ is used to initialize the derivation of superconcepts for C. The rules are sound, i.e., the conclusion subsumptions follow from the premise sub-

$$\mathbf{R_0}\ \frac{\mathsf{init}(C)}{C \sqsubseteq C} \qquad \mathbf{R}^+_\top\ \frac{\mathsf{init}(C)}{C \sqsubseteq \top} : \top \text{ occurs negatively in } \mathcal{O}$$

$$\mathbf{R}^-_\sqcap\ \frac{C \sqsubseteq D_1 \sqcap D_2}{C \sqsubseteq D_1 \quad C \sqsubseteq D_2} \qquad \mathbf{R}^+_\sqcap\ \frac{C \sqsubseteq D_1 \quad C \sqsubseteq D_2}{C \sqsubseteq D_1 \sqcap D_2} : D_1 \sqcap D_2 \text{ occurs negatively in } \mathcal{O}$$

$$\mathbf{R}^-_\exists\ \frac{C \sqsubseteq \exists R.D}{\mathsf{init}(D) \quad C \overset{R}{\to} D} \qquad \mathbf{R}^+_\exists\ \frac{E \overset{R}{\to} C \quad C \sqsubseteq D}{E \sqsubseteq \exists S.D} : \begin{array}{l} \exists S.D \text{ occurs negatively in } \mathcal{O} \\ R \sqsubseteq^*_\mathcal{O} S \end{array}$$

$$\mathbf{R}_\sqsubseteq\ \frac{C \sqsubseteq D}{C \sqsubseteq E} : D \sqsubseteq E \in \mathcal{O} \qquad \mathbf{R}_\circ\ \frac{E \overset{R_1}{\to} C \quad C \overset{R_2}{\to} D}{E \overset{S}{\to} D} : \begin{array}{l} S_1 \circ S_2 \sqsubseteq S \in \mathcal{O} \\ R_1 \sqsubseteq^*_\mathcal{O} S_1 \\ R_2 \sqsubseteq^*_\mathcal{O} S_2 \end{array}$$

Fig. 2. Inference rules for reasoning in $\mathcal{EL}^+$

sumptions and $\mathcal{O}$. The rules are complete for classification in the sense that, for each concept C and each atomic concept A occurring in $\mathcal{O}$, if $\mathcal{O} \models C \sqsubseteq A$, then $C \sqsubseteq A$ is derivable from $\mathsf{init}(C)$. Note that the axioms in $\mathcal{O}$ are never used as premises of the rules, but only as side-conditions of the rule $\mathbf{R}_\sqsubseteq$.

Example 1. Consider the ontology $\mathcal{O}$ consisting of the following axioms.

$$A \sqsubseteq \exists R.(C \sqcap D) \tag{1}$$

$$B \equiv A \sqcap \exists S.D \tag{2}$$

$$\exists S.D \sqsubseteq C \tag{3}$$

$$R \sqsubseteq S \tag{4}$$

To compute all atomic superconcepts of A, we start with the goal $\mathsf{init}(A)$ and compute all conclusions under the inference rules in Fig. 2.

$\mathsf{init}(A)$	initial assumption	(5)
$A \sqsubseteq A$	by $\mathbf{R_0}$ to (5)	(6)
$A \sqsubseteq \exists R.(C \sqcap D)$	by $\mathbf{R}_\sqsubseteq$ to (6) using (1)	(7)
$\mathsf{init}(C \sqcap D)$	by $\mathbf{R}^-_\exists$ to (7)	(8)
$A \overset{R}{\to} C \sqcap D$	by $\mathbf{R}^-_\exists$ to (7)	(9)
$C \sqcap D \sqsubseteq C \sqcap D$	by $\mathbf{R_0}$ to (8)	(10)
$C \sqcap D \sqsubseteq C$	by $\mathbf{R}^-_\sqcap$ to (10)	(11)
$C \sqcap D \sqsubseteq D$	by $\mathbf{R}^-_\sqcap$ to (10)	(12)
$A \sqsubseteq \exists S.D$	by $\mathbf{R}^+_\exists$ to (9) and (12) using (4)	(13)
$A \sqsubseteq A \sqcap \exists S.D$	by $\mathbf{R}^+_\sqcap$ to (6) and (13)	(14)
$\mathsf{init}(D)$	by $\mathbf{R}^-_\exists$ to (13)	(15)
$A \overset{S}{\to} D$	by $\mathbf{R}^-_\exists$ to (13)	(16)

$A \sqsubseteq C$	by $\mathbf{R}_{\sqsubseteq}$ to (13) using (3)	(17)
$A \sqsubseteq B$	by $\mathbf{R}_{\sqsubseteq}$ to (14) using (2)	(18)
$D \sqsubseteq D$	by $\mathbf{R}_{\mathbf{0}}$ to (15)	(19)

Since $A \sqsubseteq B$ and $A \sqsubseteq C$ have been derived but not, say, $A \sqsubseteq D$, we conclude that B and C are superconcepts of A but D is not. Incidentally, since $\mathsf{init}(D)$ was derived to satisfy the existential restriction in (13), we also computed all atomic superconcepts of D; namely, D has no nontrivial superconcepts. The application of rules $\mathbf{R}_{\exists}^{+}$ and $\mathbf{R}_{\sqcap}^{+}$ in lines (13) and (14) uses the fact that the concepts $\exists S.D$ and $A \sqcap \exists S.D$ occur negatively in $A \sqcap \exists S.D \sqsubseteq B$ which is a part of (2). Intuitively, these rules are used to "build up" the subsumption $A \sqsubseteq A \sqcap \exists S.C$, so that rule $\mathbf{R}_{\sqsubseteq}$ with side condition (2) can be applied to derive $A \sqsubseteq B$.

In order to classify an ontology $\mathcal{O}$, it is sufficient to compute the deductive closure of $\mathsf{init}(A)$ for every atomic concept A occurring in $\mathcal{O}$ using the rules in Fig. 2. Note that in this case the rules can derive only subsumptions of the form $C \sqsubseteq D$ and $C \overset{R}{\to} D$ where C, D, and R occur in $\mathcal{O}$. Therefore, the deductive closure can be computed in polynomial time.

The saturation algorithm used in ELK for computing the deductive closure is closely related to the "given clause" algorithm for saturation-based theorem proving and semi-naive (bottom-up) evaluation of logic programs. The algorithm maintains two collections of axioms: the set of *processed axioms* between which the rules have been already applied (initially empty) and the *to-do queue* of the remaining axioms (initially containing the input axioms). The algorithm repeatedly polls an axiom from the to-do queue; if the axiom is not yet in the processed set, it is moved there and the conclusions of all inferences involving this axiom and the processed axioms are added at the end of the to-do queue (regardless of whether they have been already derived).

Example 2. The derivation in Example 1 already presents the axioms in the order they are processed by the saturation algorithm. For example, after processing axiom (10), the processed set contains axioms (5)–(10), and the to-do queue contains axioms (11) and (12). The algorithm then polls axiom (11) from the queue, adds it to the processed set, and applies all inferences involving this axioms and the previously processed axioms (5)–(10). In this case, no inference rules are applicable. The algorithm then polls the next axiom (12) from the queue, adds it to the processed set, and applies all inferences involving this axioms and the previously processed axioms (5)–(11). In this case, rule $\mathbf{R}_{\exists}^{+}$ is applicable to the premises (9) and (10), so the conclusion (13) is added to the to-do queue.

4 Implementation of Reasoning

In this section, we give details of the three main phases of the classification algorithm that is the core of ELK's reasoning module (see Fig. 1): indexing, saturation, and taxonomy construction.

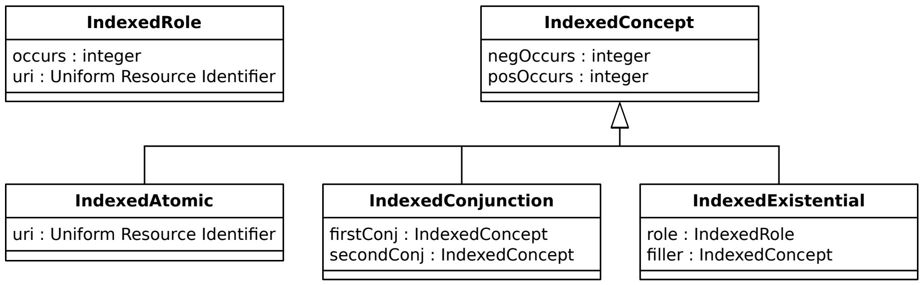

Fig. 3. Classes for indexing objects

4.1 Indexing

The task of the indexing phase is to build an internal representation of the input ontology $\mathcal{O}$ that is suitable for checking the side conditions of the inference rules efficiently. As illustrated in Fig. 1, the input that is used for indexing is an ontology in ELK's lightweight representation of OWL expressions and axioms, which may have been obtained from OWL FSS files, OWL API objects, or any other source. While ELK's representation of OWL faithfully captures the structure of the ontology in terms of OWL language features, the indexed ontology represents only the description logic semantics that is essential for applying the inference rules. For this purpose, a system of *indexed objects* is used to represent DL (rather than OWL) role and concept expressions. These objects are the basis for all further internal index structures.

The type hierarchy of indexed objects is shown in Fig. 3; it closely follows the recursive definition of $\mathcal{EL}$ expressions. The top level classes IndexedRole and IndexedConcept are used to represent roles and concepts respectively. For each indexed role, the field occurs stores the number of times this role occurs in the ontology $\mathcal{O}$, and the field uri contains the URI that identifies the role. For indexed concepts we separately keep track of the number of their negative and positive occurrences in the fields negOccurs and posOccurs. The different types of $\mathcal{EL}$ concepts are represented by the corresponding subclasses of IndexedConcept from Fig. 3. For example, a conjunction $C \sqcap D$ is represented by an object of type IndexedConjunction, and its fields firstConj and secondConj point to the indexed objects that represent the concepts C and D, respectively. In addition, there is a distinguished instance top of IndexedConcept that represents the concept $\top$.

Recall that OWL supports conjunctions of arbitrary arity. For efficient indexing, ELK converts these to series of nested binary conjunctions, e.g., the concept $A \sqcap B \sqcap C \sqcap D$ would be represented as $((A \sqcap B) \sqcap C) \sqcap D$. A similar transformation is used when restricting to binary role composition axioms below.

Apart from creating the indexed objects, indexing also constructs data structures for efficient look up of side conditions of the inference rules. Specifically, the following tables (sets of tuples) of indexed objects are constructed. To simplify the presentation, in the following we ignore the distinction between DL expressions and their indexed representations.

$$\begin{aligned}
\mathsf{negConjs} &= \{\langle C, D, C \sqcap D\rangle \mid C \sqcap D \text{ occurs negatively in } \mathcal{O}\}\\
\mathsf{negExis} &= \{\langle R, C, \exists R.C\rangle \mid \exists R.C \text{ occurs negatively in } \mathcal{O}\}\\
\mathsf{conceptIncs} &= \{\langle C, D\rangle \mid C \sqsubseteq D \in \mathcal{O}\}\\
\mathsf{roleIncs} &= \{\langle R, S\rangle \mid R \sqsubseteq S \in \mathcal{O}\}\\
\mathsf{roleComps} &= \{\langle R_1, R_2, S\rangle \mid R_1 \circ R_2 \sqsubseteq S \in \mathcal{O}\}
\end{aligned}$$

Example 3. Consider the ontology $\mathcal{O}$ from Example 1. Its index contains the following indexed objects with occurrence counts as indicated.

IndexedRole	R	S
occurs	2	4

IndexedConcept	$\top$	A	B	C	D	$C \sqcap D$	$\exists R.(C \sqcap D)$	$\exists S.D$	$A \sqcap \exists S.D$
negOccurs	0	2	1	0	2	0	0	2	1
posOccurs	0	1	1	2	2	1	1	1	1

Recall that we treat the equivalence axiom (2) as two individual concept inclusion axioms, so, e.g., the occurrence count of S is 4, not 3. The look-up tables contain the following tuples.

$$\begin{aligned}
\mathsf{negConjs} &= \{\langle A, \exists S.D, A \sqcap \exists S.D\rangle\}\\
\mathsf{negExis} &= \{\langle S, D, \exists S.D\rangle\}\\
\mathsf{conceptIncs} &= \{\langle A, \exists R.(C \sqcap D)\rangle, \langle B, A \sqcap \exists S.D\rangle, \langle A \sqcap \exists S.D, B\rangle, \langle \exists S.D, C\rangle\}\\
\mathsf{roleIncs} &= \{\langle R, S\rangle\}\\
\mathsf{roleComps} &= \emptyset
\end{aligned}$$

To illustrate how the index is accessed during the saturation phase, consider the point in the derivation from Example 1 when the saturation algorithm processes the axiom $A \sqsubseteq \exists S.D$ in line (13). To apply rule $\mathbf{R}^{+}_{\sqcap}$ to this axiom, the algorithm iterates over those tuples in negConjs whose first or second component is $\exists S.D$ to find possible ways of satisfying the side condition. Since negConjs contains $\langle A, \exists S.D, A \sqcap \exists S.D\rangle$, the algorithm checks if the set of processed axioms contains $A \sqsubseteq A$, which can be used as the first premise of $\mathbf{R}^{+}_{\sqcap}$. Since this is the case, the conclusion $A \sqsubseteq A \sqcap \exists S.D$ is added to the to-do queue. Note that (a pointer to) the indexed conjunction $A \sqcap \exists S.D$ used in the conclusion can be taken directly from the table negConjs. This illustrates that conclusions of the inference rules can be constructed by simply following the pointers in the index, and no new indexed object need to be created during the saturation phase.

Indexing is a lightweight task that can be performed by a single recursive traversal through the structure of each axiom in the ontology. Since it can consider one axiom at a time, it can be started even before the whole ontology is known to the reasoner. In ELK, indexing is executed in a second thread in parallel to loading of axioms. Moreover, if new axioms are added to the ontology, it is sufficient to index the new axioms without having to reload the whole ontology. Since we keep the exact counts of negative and positive the occurrences of concepts, the same also applies to deletions of axioms.

Example 4. Consider a deletion of the axiom $B \equiv A \sqcap \exists S.D$ from the ontology in Example 1. Firstly, the tuples $\langle B, A \sqcap \exists S.D \rangle$ and $\langle A \sqcap \exists S.D, B \rangle$ are removed from the table $\mathsf{conceptIncs}$. Secondly, both the negative and the positive occurrence counts of B, $A \sqcap \exists S.D$, A, and $\exists S.D$ are decremented, and the occurrence count of S is decreased by 2. Since the negative occurrence count of $A \sqcap \exists S.D$ drops to zero, the tuple $\langle A, \exists S.D, A \sqcap \exists S.D \rangle$ is removed from the table $\mathsf{negConjs}$. On the other hand, the negative occurrence count of $\exists S.D$ remains nonzero (the concept still occurs negatively in (3)), so the table $\mathsf{negExis}$ remains unchanged. Finally, since there are no more occurrences of B and $A \sqcap \exists S.D$, the corresponding indexed objects are deleted.

4.2 Saturation of Roles

In this phase we precompute the reflexive transitive closure $\sqsubseteq^*_{\mathcal{O}}$ of the role inclusion axioms of $\mathcal{O}$, which is needed for efficient application of rules $\mathbf{R}^+_\exists$ and $\mathbf{R}_\circ$, and store it in a table called hier (for role hierarchy).

$$\mathsf{hier} = \{\langle R, S \rangle \mid R \sqsubseteq^*_{\mathcal{O}} S\}.$$

Since the number of roles in real-world ontologies is usually much smaller than the number of concepts, any reasonable algorithm for computing the transitive closure can be used with no significant impact on the overall performance.

Example 5. For the ontology $\mathcal{O}$ from Example 1, we have

$$\mathsf{hier} = \{\langle R, R \rangle, \langle R, S \rangle, \langle S, S \rangle\}.$$

4.3 Saturation of Concepts

In this phase we compute the deductive closure under the inference rules in Fig. 2. Recall that the inference rule operate with three types of axioms $\mathsf{init}(C)$, $C \sqsubseteq D$, and $C \overset{R}{\to} D$. We store such axioms in three separate tables init, subs, links as follows.

axiom	internal representation
$\mathsf{init}(C)$	$C \in \mathsf{init}$
$C \sqsubseteq D$	$\langle C, D \rangle \in \mathsf{subs}$
$C \overset{R}{\to} D$	$\langle C, R, D \rangle \in \mathsf{links}$

Before we present the details of the saturation algorithm, we first compile the inference rule from Fig. 2 to operate directly with tables and indexed objects instead of DL expressions. The compiled rules are shown in Fig. 4, where, for a table T and a tuple x, we use the notation $T(x)$ as a shorthand for $x \in T$.

$\mathsf{R_0}$: If $\mathsf{init}(C)$,
then $\mathsf{subs}(C, C)$.

$\mathsf{R}^{+}_{\top}$: If $\mathsf{init}(C) \wedge \mathsf{top.negOccurs} > 0$,
then $\mathsf{subs}(C, \mathsf{top})$.

$\mathsf{R}^{-}_{\sqcap}$: If $\mathsf{subs}(C, D) \wedge D$ **instanceOf** $\mathsf{IndexedConjunction}$,
then $\mathsf{subs}(C, D.\mathsf{firstConj})$ and $\mathsf{subs}(C, D.\mathsf{secondConj})$.

$\mathsf{R}^{+}_{\sqcap}$: If $\mathsf{subs}(C, D_1) \wedge \mathsf{subs}(C, D_2) \wedge \mathsf{negConjs}(D_1, D_2, D)$,
then $\mathsf{subs}(C, D)$.

$\mathsf{R}^{-}_{\exists}$: If $\mathsf{subs}(C, D) \wedge D$ **instanceOf** $\mathsf{IndexedExistential}$,
then $\mathsf{init}(D.\mathsf{filler})$ and $\mathsf{links}(C, D.\mathsf{role}, D.\mathsf{filler})$.

$\mathsf{R}^{+}_{\exists}$: If $\mathsf{links}(E, R, C) \wedge \mathsf{subs}(C, D) \wedge \mathsf{negExis}(S, D, F) \wedge \mathsf{hier}(R, S)$,
then $\mathsf{subs}(E, F)$.

$\mathsf{R}_{\sqsubseteq}$: If $\mathsf{subs}(C, D) \wedge \mathsf{conceptIncs}(D, E)$,
then $\mathsf{subs}(C, E)$.

$\mathsf{R}_{\circ}$: If $\mathsf{links}(E, R_1, C) \wedge \mathsf{links}(C, R_2, D) \wedge \mathsf{roleComps}(S_1, S_2, S) \wedge$
$\mathsf{hier}(R_1, S_1) \wedge \mathsf{hier}(R_2, S_2)$,
then $\mathsf{links}(E, S, D)$.

Fig. 4. Compiled inference rules

As explained in Section 3, the algorithm does not actually put the conclusions of the inference rules directly into the corresponding tables. Instead, it maintains a queue todo of newly derived axioms for which the inference rules are yet to be applied, repeatedly polls an axiom from todo, stores it in the corresponding table if not already present, and applies all inferences involving this axiom and the earlier processed axioms, whose conclusions are put to todo.

The main body of the saturation algorithm is shown in Algorithm 1: for classification, it initializes the computation with $\mathsf{init}(A)$ for all atomic concepts occurring in the ontology, and then repeatedly processes axioms from todo until the queue is empty. The implementation of the function `process(axiom)` for each of the three axiom types is shown in Algorithm 2.

Algorithm 1: Main body of the saturation algorithm

```
1 for each IndexedAtomic $A$ do
2     todo.add(init($A$));
3 while todo $\neq \emptyset$ do
4     axiom $\leftarrow$ todo.poll();
5     process(axiom);
```

Algorithm 2: process(axiom)

```
process(init($C$)):
if init.add($C$) then
    todo.add($C \sqsubseteq C$);                                  // rule $\mathbf{R_0}$
    if top.negOccurs $> 0$ then
        todo.add($C \sqsubseteq$ top);                            // rule $\mathbf{R}^+_\top$

process($C \sqsubseteq D$):
if subs.add($\langle C, D \rangle$) then
    if $D$ instanceOf IndexedConjunction then
        todo.add($C \sqsubseteq D$.firstConj);
        todo.add($C \sqsubseteq D$.secondConj);                   // rule $\mathbf{R}^-_\sqcap$
    if $D$ instanceOf IndexedExistential then
        todo.add(init($D$.filler));
        todo.add($C \overset{D.\mathsf{role}}{\rightarrow} D$.filler);   // rule $\mathbf{R}^-_\exists$
    for each $D_2, E$ with subs($C, D_2$) $\wedge$ negConjs($D, D_2, E$) do
        todo.add($C \sqsubseteq E$);                              // rule $\mathbf{R}^+_\sqcap$
    for each $D_1, E$ with subs($C, D_1$) $\wedge$ negConjs($D_1, D, E$) do
        todo.add($C \sqsubseteq E$);                              // rule $\mathbf{R}^+_\sqcap$
    for each $E, F, R, S$ with links($E, R, C$) $\wedge$ negExis($S, D, F$) $\wedge$ hier($R, S$) do
        todo.add($E \sqsubseteq F$);                              // rule $\mathbf{R}^+_\exists$
    for each $E$ with conceptIncs($D, E$) do
        todo.add($C \sqsubseteq E$);                              // rule $\mathbf{R}_\sqsubseteq$

process($E \overset{R}{\rightarrow} C$):
if links.add($\langle E, R, C \rangle$) then
    for each $D, F, S$ with subs($C, D$) $\wedge$ negExis($S, D, F$) $\wedge$ hier($R, S$) do
        todo.add($E \sqsubseteq F$);                              // rule $\mathbf{R}^+_\exists$
    for each $D, R_2, S_1, S_2, S$ with links($C, R_2, D$) $\wedge$ roleComps($S_1, S_2, S$) $\wedge$
    hier($R, S_1$) $\wedge$ hier($R_2, S_2$) do
        todo.add($E \overset{S}{\rightarrow} D$);                 // rule $\mathbf{R}_\circ$
    for each $D, R_1, S_1, S_2, S$ with links($D, R_1, E$) $\wedge$ roleComps($S_1, S_2, S$) $\wedge$
    hier($R_1, S_1$) $\wedge$ hier($R, S_2$) do
        todo.add($D \overset{S}{\rightarrow} C$);                 // rule $\mathbf{R}_\circ$
```

Note that Algorithm 2 often iterates over joins of tables; optimizing such iterations is essential for achieving efficiency of the algorithm. We will discuss several possible optimizations in Section 5.1.

Example 6. Here we show the result of applying the saturation algorithm to the ontology from Example 1. The saturation includes (a representation of) all the axioms that were derived in Example 1, together with those that are derivable from $\mathsf{init}(B)$ and $\mathsf{init}(C)$.

$$\begin{aligned}
\mathsf{init} &= \{A, B, C, D, C \sqcap D\} \\
\mathsf{subs} &= \{\langle A, A\rangle, \langle A, \exists R.(C \sqcap D)\rangle, \langle A, \exists S.D\rangle, \langle A, A \sqcap \exists S.D\rangle, \langle A, C\rangle, \langle A, B\rangle \\
&\quad\ \langle B, B\rangle, \langle B, A \sqcap \exists S.D\rangle, \langle B, A\rangle, \langle B, \exists S.D\rangle, \langle B, \exists R.(C \sqcap D)\rangle, \langle B, C\rangle \\
&\quad\ \langle C, C\rangle, \langle D, D\rangle, \langle C \sqcap D, C \sqcap D\rangle, \langle C \sqcap D, C\rangle, \langle C \sqcap D, D\rangle\} \\
\mathsf{links} &= \{\langle A, R, C \sqcap D\rangle, \langle A, S, D\rangle, \langle B, S, D\rangle, \langle B, R, C \sqcap D\rangle\}
\end{aligned}$$

4.4 Taxonomy Construction

The concept saturation phase computes the full transitively closed subsumption relation. However, the expected output of classification is a *taxonomy* which only contains direct subsumptions between nodes representing equivalence classes of atomic concepts. Therefore, the computed subsumptions between atomic concepts must be transitively reduced.

In the first step, we discard all subsumptions derived by the saturation algorithm that involve non-atomic concepts. Thus, in the remainder of this section, we can assume that all concepts are atomic.

Algorithm 3: Naive Transitive Reduction

```
1 for each $C$ with subs($A, C$) do
2     isDirect ← true;
3     for each $B$ with subs($A, B$) do
4         if $B \neq C$ and subs($B, C$) then
5             isDirect ← false;
6     if isDirect then
7         $A$.directSuperConcepts.add($C$)
```

A naive solution for computing the direct superconcepts of A is shown in Algorithm 3. The algorithm iterates over all superconcepts C of A, and for each of them checks if another superconcept B of A exists with $A \sqsubseteq B \sqsubseteq C$. If no such B exists, then C is a direct superconcept of A. Apart from the fact this approach does not work as expected when A has two equivalent superconcepts, in which case none of them would be found as direct, the algorithm is inefficient because it performs two nested iterations over the superconcepts of A. In realistic ontologies, the number of all superconcepts of A can be sizeable, while the number of direct superconcepts is usually much smaller, often just one. A more efficient algorithm would take advantage of this and perform the inner iteration only over the set of direct superconcepts of A that have been found so far, as shown in Algorithm 4. Given A, the algorithm computes two sets A.equivalentConcepts and A.directSuperConcepts. The first set contains all concepts that are equivalent to A, including A itself. The second set contains exactly one element from each equivalence class of direct superconcepts of A. Note that it is safe to execute Algorithm 4 in parallel for multiple concepts A.

Algorithm 4: Better Transitive Reduction

```
for each $C$ with subs($A, C$) do
    if subs($C, A$) then
        $A$.equivalentConcepts.add($C$);
    else
        isDirect ← true;
        for $B \in A$.directSuperConcepts do
            if subs($B, C$) then
                isDirect ← false;
                break;
            if subs($C, B$) then
                $A$.directSuperConcepts.remove($B$);
        if isDirect then
            $A$.directSuperConcepts.add($C$);
```

Having computed A.equivalentConcepts and A.directSuperConcepts for each A, the construction of the taxonomy is straightforward. We introduce one taxonomy node for each distinct class of equivalent concepts, and connect the nodes according to the direct superconcepts relation. Care has to be taken to put the top and the bottom node in their proper positions, even if $\top$ or $\bot$ do not occur in the ontology.

Example 7. Consider the ontology from Example 1 and its saturation from Example 6. Projecting the subsumptions to atomic concepts leaves us with

$$Subs = \{\langle A, A\rangle, \langle A, B\rangle, \langle A, C\rangle, \langle B, A\rangle, \langle B, B\rangle, \langle B, C\rangle, \langle C, C\rangle, \langle D, D\rangle\},$$

from which Algorithm 4 computes

$$\begin{aligned} &A.\mathsf{equivalentConcepts} = \{A, B\}, && A.\mathsf{directSuperConcepts} = \{C\}, \\ &B.\mathsf{equivalentConcepts} = \{A, B\}, && B.\mathsf{directSuperConcepts} = \{C\}, \\ &C.\mathsf{equivalentConcepts} = \{C\}, && C.\mathsf{directSuperConcepts} = \emptyset, \\ &D.\mathsf{equivalentConcepts} = \{D\}, && D.\mathsf{directSuperConcepts} = \emptyset. \end{aligned}$$

The resulting taxonomy is shown in Fig. 5.

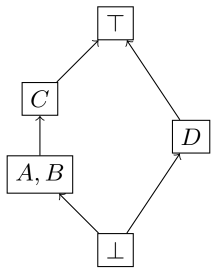

Fig. 5. Taxonomy for the ontology from Example 1

5 Further Optimizations

5.1 Join Iteration

Iterations over joins of tables are used extensively in Algorithm 2 to retrieve all matching side-conditions (and second premises) of inference rules. We optimize such retrievals by iterating over smaller tables and caching partial joins.

We illustrate the main ideas on the conjunction rule $\mathsf{R}^+_\sqcap$ implemented in Algorithm 2 in lines 14–17. To apply $\mathsf{R}^+_\sqcap$ with the given axiom $C \sqsubseteq D$ as the first premise, the for loop in line 14 iterates over all indexed concepts D_2 and E such that $\mathsf{subs}(C, D_2) \wedge \mathsf{negConjs}(D, D_2, E)$, i.e., such that the subsumption $C \sqsubseteq D_2$ has already been processed and the conjunction $E = D \sqcap D_2$ occurs negatively in the ontology.

A simple solution would be to first retrieve all D_2 with $\mathsf{subs}(C, D_2)$ and then for each of them retrieve all (if any) E with $\mathsf{negConjs}(D, D_2, E)$. With suitable hash indexes over the tables, such retrievals can be executed efficiently. Another possibility is to perform the retrievals in the opposite order: first retrieve all pairs of D_2 and E with $\mathsf{negConjs}(D, D_2, E)$ and then for each of them check if $\mathsf{subs}(C, D_2)$. Since most concepts occur only in very few negative conjunctions, this order often requires fewer retrievals than the former. Of course, a better solution is to analyse the number of required retrievals upfront and then dynamically select the order which requires fewer retrievals.

The above approach already performs reasonably well in practice, but it can be optimized further. Although most concepts occur only in few negative conjunctions, it is common that an ontology contains a small number of (usually very general) concepts that occur in many conjunctions. For example, there is one concept in SNOMED CT that occurs in about 1,400 negative conjunctions. For definitiveness, let us consider a situation where one general concept A occurs in conjunctions $A \sqcap B_i$ for many specific concepts B_i. Whenever $C \sqsubseteq A$ is derived for some C, which happens often because A is general, for the application of rule $\mathsf{R}^+_\sqcap$ the algorithm needs to retrieve all D_2 (and the corresponding E) from the intersection of the set of previously derived superconcepts of C and the set $\{B_i\}$. While both of these sets can be large, their intersection is likely to be small because all B_i are specific and therefore do not occur often as superconcepts. In such case, iterating over either of the two sets is inefficient, and a better solution might be to have the intersection precomputed and update it whenever some axiom $C \sqsubseteq B_i$ is derived. More specifically, the algorithm will work as follows. When processing an axiom $C \sqsubseteq B_i$, the algorithm checks if $C \sqsubseteq A$ has already been derived. If yes, $C \sqsubseteq A \sqcap B_i$ is derived as usual. If no, however, then the partially matched *implication axiom* $C \sqsubseteq (A \to A \sqcap B_i)$ is stored for future reference (in practice, the implication is stored as a triple $\langle C, A, A \sqcap C_i \rangle$ in a new table impl). When processing an axiom $C \sqsubseteq A$, the algorithm implements rule $\mathsf{R}^+_\sqcap$ simply by retrieving all implications of the form $C \sqsubseteq (A \to E)$ (i.e., all E with $\mathsf{impl}(C, A, E)$) and derives the conclusions $C \sqsubseteq E$.

Note that deriving implication axioms is most useful when there is an asymmetry between the conjuncts as above: the C_i's occur rarely, therefore the algorithm derives few implications $C \sqsubseteq (A \to A \sqcap C_i)$, and the algorithm saves many long iterations when applying rule $\mathsf{R}^+_\sqcap$ to axioms $C \sqsubseteq A$. On the other hand, when processing axioms $C \sqsubseteq A$, it is wasteful to derive the implications $C \sqsubseteq (C_i \to A \sqcap C_i)$. Since A occurs often and there are many C_i's, this would derive a large number of implication axioms, but because C_i's occur rarely, only few of these implications will be fire.

Similar consideration apply to rule $\mathsf{R}^+_\exists$. Let us ignore the role hierarchy for the moment. In that case, to apply rule $\mathsf{R}^+_\exists$ to an axiom $E \overset{R}{\to} C$ as the first premise, the for loop in line 24 in Algorithm 2 iterates over all D and F such that $\mathsf{subs}(C, D)$ and $\mathsf{negExis}(R, D, F)$, i.e., such that the subsumption $C \sqsubseteq D$ has already been processed and the existential $F = \exists R.D$ occurs negatively in the ontology. This is analogous to the previously discussed application of rule $\mathsf{R}^+_\sqcap$, only using table $\mathsf{negExis}$ instead of $\mathsf{negConjs}$. Here the asymmetry between roles and concepts is much more common than in the case of conjuncts: it is usually the case that for a role R there are many negative existentials $\exists R.C_i$, but for a concept C there are only few negative existentials $\exists R_i.C$. Therefore, this suggests that it might be better to precompute the join $\mathsf{subs}(C, D) \bowtie_D \mathsf{negExis}(R, D, F)$. The algorithm works as follows. When processing an axiom $C \sqsubseteq D$, the algorithm derives the *propagation axiom* $C \sqsubseteq (R \to F)$ for each R and F such that $\mathsf{negExis}(R, D, F)$ (the propagation will be stored as the triple $\langle C, R, F \rangle$ in a new table prop). When processing an axiom $E \overset{R}{\to} C$, the algorithm implements rule

$\mathsf{R}^+_\exists$ by retrieving all propagations of the form $C \sqsubseteq (R \to F)$ (i.e., all F with $\mathsf{prop}(C, R, F)$), for which it derives the conclusions $E \sqsubseteq F$. This corresponds essentially to the inference rule $\mathsf{R_H}$ from [2], which can be stated as follows:

$$\mathsf{R_H}\ \frac{E \overset{R}{\to} C \quad C \sqsubseteq (R \to F)}{E \sqsubseteq F}.$$

Adding the role hierarchy introduces another join in the application of rule $\mathsf{R}^+_\exists$ as in Fig. 4. There are several possible of ways dealing with the additional join, we mention a few of them. One possibility (which is currently implemented in ELK) is to add an intermediate iteration over the role hierarchy when executing rule $\mathsf{R_H}$ as follows:

$$\mathsf{R_H}\ \frac{E \overset{R}{\to} C \quad C \sqsubseteq (S \to F)}{E \sqsubseteq F} : R \sqsubseteq^*_{\mathcal{O}} S.$$

Another possibility it to precompute the join $\mathsf{links} \bowtie \mathsf{hier}$, which essentially corresponds to expanding the axioms $E \overset{R}{\to} C$ under the role hierarchy as follows:

$$\frac{E \overset{R}{\to} C}{E \overset{S}{\to} C} : R \sqsubseteq^*_{\mathcal{O}} S.$$

Yet another possibility is to precompute the join of $\mathsf{prop} \bowtie \mathsf{hier}$, which corresponds to expanding propagation axioms under the role hierarchy as follows:

$$\frac{C \sqsubseteq (S \to F)}{C \sqsubseteq (R \to F)} : R \sqsubseteq^*_{\mathcal{O}} S.$$

There is a trade-off between time and space in the above approaches: expansion of axioms under the role hierarchy requires more space, but it eliminates the need to iterate over the role hierarchy in the application of rule $\mathsf{R_H}$.

5.2 Concurrent Saturation

The saturation phase can be further sped up by processing multiple axioms concurrently. The basic idea is to employ several workers (running in different threads) to execute the loop in lines 3–5 of Algorithm 1 at the same time. To make this possible, the to-do queue has to support concurrent insertions and retrievals of elements from different threads. Fortunately, standard Java libraries already provide efficient implementations of such concurrent queues.

More problematically, the shared tables accessed from `process(axiom)` of Algorithm 2 must be thread-safe. For example, several workers might try to insert subsumptions into the table subs line 7 at the same time, and, moreover, other workers might already be iterating over subs in lines 14 or 24 at this time. The procedure must behave correctly in all such cases.

A simple solution would be to use locks on the three tables init, subs, and links that are written to during the saturation phase. This would ensure that there is always at most one worker accessing these tables at one time, but it would largely defeat the purpose of concurrent computation since the workers will have to wait for each other in order to proceed.

We propose a solution which does not use locks. Instead of having just three tables init, subs, and links, the idea is to implement each of these tables as a collection of a large number of subtables, and ensure that no two workers need to access the same subtable at the same time.

To this end, we assign the axioms that are derived during the saturation phase into *contexts*. We have one context for each initialized concept C (for which $\mathsf{init}(C)$ has been derived). An axiom of the form $\mathsf{init}(C)$, $C \sqsubseteq D$, or $E \overset{R}{\to} C$ is assigned to the context of C. Additionally, in the presence of role composition axioms, the axiom $E \overset{R}{\to} C$ is also assigned to the context of E. The assignment of axioms to contexts satisfies the important property that whenever a binary inference rule ($\mathbf{R}^+_\sqcap$, $\mathbf{R}^+_\exists$, and $\mathbf{R}_\circ$) is applicable to a pair of premises, the two premises belong to the same context. Therefore, when a worker processes an axiom in one context, it only needs to access those processed axioms that are in the same context, so multiple workers can independently process axioms in different contexts at the same time

To ensure that no two workers are concurrently processing axioms in the same context, the to-do queue is split into a two-level hierarchy of queues: each context of C maintains a local queue $C.\mathsf{todo}$ of to-do axioms that are assigned to the context of C, and there is a global queue of *active contexts* whose to-do queues are nonempty. Using concurrency techniques, such as Boolean flags with atomic compare-and-set operations, the queue of active contexts is kept duplicate free. Each worker then repeatedly polls an active context C from the queue and processes all axioms in $C.\mathsf{todo}$. More details can be found in [2].

The following table shows the internal representation of the different types of derivable axioms in the refined saturation algorithm with contexts. For example, instead of representing a processed subsumption $C \sqsubseteq D$ by the pair $\langle C, D\rangle$ in the global table subs, the indexed concept D will be stored in the set $C.\mathsf{superConcepts}$ which is local to the context of C. The second column of the table shows how the axioms are stored in the to-do queues. Instead of actually deriving axioms of the form $\mathsf{init}(C)$, the algorithm uses a method `initializeContext`(C) to initialize the context of C and apply rules $\mathbf{R_0}$ and $\mathbf{R}^+_\top$. See Algorithm 5 for details.

axiom	to-do representation	processed representation
$\mathsf{init}(C)$		$C.\mathsf{isInitialized} == \mathbf{true}$
$C \sqsubseteq D$	$\mathsf{SuperConcept}(D) \in C.\mathsf{todo}$	$D \in C.\mathsf{superConcepts}$
$E \overset{R}{\to} C$	$\mathsf{Predecessor}(R,E) \in C.\mathsf{todo}$	$\langle R, E\rangle \in C.\mathsf{predecessors}$
	$\mathsf{Successor}(R,C) \in E.\mathsf{todo}$	$\langle R, C\rangle \in E.\mathsf{successors}$

Note that storing an axiom $E \overset{R}{\to} C$ in the table $E.\mathsf{successors}$ is only needed in the presence of role compositions for the execution of rule $\mathbf{R}_\circ$ in line 27.

5.3 Composition-Decomposition Optimizations

In this section we present an optimization of the inference system that allows us to omit certain applications of rules $\mathbf{R}^-_\sqcap$ and $\mathbf{R}^-_\exists$ without losing completeness.

Algorithm 5: C.process(axiom)

```
initializeContext($C$):
if $C$.isInitialized == false then
    $C$.isInitialized ← true;
    $C$.todo.add(SuperConcept($C$));                                  // rule $\mathbf{R_0}$
    if top.negOccurs > 0 then
        $C$.todo.add(SuperConcept(top));                              // rule $\mathbf{R}^{+}_{\top}$

process(SuperConcept($D$)):
if $C$.superConcepts.add($D$) then
    if $D$ instanceOf IndexedConjunction then
        $C$.todo.add(SuperConcept($D$.firstConj));
        $C$.todo.add(SuperConcept($D$.secondConj));                   // rule $\mathbf{R}^{-}_{\sqcap}$
    if $D$ instanceOf IndexedExistential then
        initializeContext($D$.filler);
        $D$.filler.todo.add(Predecessor($D$.role, $C$));              // rule $\mathbf{R}^{-}_{\exists}$
    for each $D_2, E$ with $C$.superConcepts($D_2$) $\wedge$ negConjs($D, D_2, E$) do
        $C$.todo.add(SuperConcept($E$));                              // rule $\mathbf{R}^{+}_{\sqcap}$
    for each $D_1, E$ with $C$.superConcepts($D_1$) $\wedge$ negConjs($D_1, D, E$) do
        $C$.todo.add(SuperConcept($E$));                              // rule $\mathbf{R}^{+}_{\sqcap}$
    for each $E, F, R, S$ with $C$.predecessors($R, E$) $\wedge$ negExis($S, D, F$) $\wedge$
    hier($R, S$) do
        $E$.todo.add(SuperConcept($F$));                              // rule $\mathbf{R}^{+}_{\exists}$
    for each $E$ with conceptIncs($D, E$) do
        $C$.todo.add(SuperConcept($E$));                              // rule $\mathbf{R}_{\sqsubseteq}$

$C$.process(Predecessor($R, E$)):
if $C$.predecessors.add($\langle R, E\rangle$) then
    for each $D, F, S$ with $C$.superConcepts($D$) $\wedge$ negExis($S, D, F$) $\wedge$ hier($R, S$)
    do
        $E$.todo.add(SuperConcept($F$));                              // rule $\mathbf{R}^{+}_{\exists}$
    for each $D, R_2, S_1, S_2, S$ with $C$.successors($R_2, D$) $\wedge$ roleComps($S_1, S_2, S$)
    $\wedge$ hier($R, S_1$) $\wedge$ hier($R_2, S_2$) do
        $D$.todo.add(predecessors($S, E$));                           // rule $\mathbf{R}_{\circ}$
    /* the following line is only needed in the presence of role
       composition axioms                                                          */
    $E$.todo.add(Successor($R, C$));

$C$.process(Successor($R, D$)):
if $C$.successors.add($\langle R, D\rangle$) then
    for each $E, R_1, S_1, S_2, S$ with $C$.predecessors($R_1, E$) $\wedge$ roleComps($S_1, S_2, S$)
    $\wedge$ hier($R_1, S_1$) $\wedge$ hier($R, S_2$) do
        $D$.todo.add(Predecessor($S, E$));                            // rule $\mathbf{R}_{\circ}$
```

Consider again the derivation in Example 1. Rules $\mathsf{R}_\exists^+$ in step (13) and $\mathsf{R}_\sqcap^+$ in step (14) are used to explicitly "build up" the complex subsumption $A \sqsubseteq A \sqcap \exists S.C$ from the already known facts $A \sqsubseteq A$, $A \sqsubseteq \exists R.(C \sqcap D)$, $R \sqsubseteq_\mathcal{O}^* S$, and $(C \sqcap D) \sqsubseteq C$, so that rule $\mathsf{R}_\sqsubseteq$ can be applied in step (18) to this subsumption to derive the new consequence $A \sqsubseteq B$.

The above example illustrates the general behaviour of the inference system. Rules $\mathsf{R}_\sqcap^+$ and $\mathsf{R}_\exists^+$ are applied to *compose* already derived facts into complex subsumptions $C \sqsubseteq D$ with D occurring negatively in $\mathcal{O}$. When the point is reached that D is the entire left-hand side of some axiom $D \sqsubseteq E$ in $\mathcal{O}$, then rule $\mathsf{R}_\sqsubseteq$ is applied to derive the subsumption $C \sqsubseteq E$. If E is a complex concept, then rules $\mathsf{R}_\sqcap^-$ and $\mathsf{R}_\exists^-$ are applied to repeatedly *decompose* the newly found subsumption $C \sqsubseteq E$ into parts, and the whole process repeats. This motivates that it is not necessary to apply the decomposition rules to decompose the subsumptions that were composed by the composition rules.

In practice, this amounts to the following two optimizations.

$O_\sqcap$ It is not necessary to apply rule $\mathsf{R}_\sqcap^-$ to a subsumption $C \sqsubseteq D_1 \sqcap D_2$ that was derived by rule $\mathsf{R}_\sqcap^+$.

$O_\exists$ It is not necessary to apply rule $\mathsf{R}_\exists^-$ to a subsumption $C \sqsubseteq \exists S.E$ that was derived by rule $\mathsf{R}_\exists^+$.

Optimization $O_\sqcap$ is rather trivial: if $C \sqsubseteq D_1 \sqcap D_2$ was derived by rule $\mathsf{R}_\sqcap^+$, then both $C \sqsubseteq D_1$ and $C \sqsubseteq D_2$ must have been derived earlier, so it is not necessary to derive them again in rule $\mathsf{R}_\sqcap^-$. Even then, avoiding the actual application of the rule can still offer some speedup of the algorithm.

Optimization $O_\exists$ is more interesting: if $C \sqsubseteq \exists S.E$ was derived by rule $\mathsf{R}_\exists^+$, then some $C \overset{R}{\to} D$ with $R \sqsubseteq_\mathcal{O}^* S$ and $D \sqsubseteq E$ must have been derived earlier, but not necessary the conclusion $C \overset{S}{\to} E$ of rule $\mathsf{R}_\exists^-$.

Correctness of optimization $O_\exists$ can perhaps be understood better by considering the standard construction of canonical models, where an axiom $C \overset{R}{\to} D$ indicates that the canonical instance of C is connected by the role R to the canonical instance of D. Given $C \sqsubseteq \exists S.E$, rule $\mathsf{R}_\exists^-$ is used to satisfy the existential restriction: deriving $C \overset{S}{\to} E$ ensures that the canonical instance of C has an S-successor of type E. However, this is also guaranteed to hold if we already have $C \overset{R}{\to} D$ with $R \sqsubseteq_\mathcal{O}^* S$ and $D \sqsubseteq E$, in which case the canonical instance of D will be the required S-successor of type E. Therefore, the conclusion $C \overset{S}{\to} E$ of rule $\mathsf{R}_\exists-$ is redundant.

Example 8. Using optimization $O_\exists$, the application of rule $\mathsf{R}_\exists^-$ to axiom (13) in Example 1 is redundant. This avoids deriving axioms (15), (16), and (19), and, in particular, even avoids the need to introduce the context of D.

References

1. Baader, F., Lutz, C., Suntisrivaraporn, B.: CEL—a polynomial-time reasoner for life science ontologies. In: Proc. 3rd Int. Joint Conf. on Automated Reasoning (IJCAR'06). LNCS, vol. 4130, pp. 287–291. Springer (2006)

2. Kazakov, Y., Krötzsch, M., Simančík, F.: Concurrent classification of $\mathcal{EL}$ ontologies. In: Proc. 10th Int. Semantic Web Conf. (ISWC'11). LNCS, vol. 7032. Springer (2011)
3. Kazakov, Y., Krötzsch, M., Simančík, F.: Practical reasoning with nominals in the $\mathcal{EL}$ family of description logics. Tech. rep., University of Oxford (2011), available from http://code.google.com/p/elk-reasoner/wiki/Publications
4. Kazakov, Y., Krötzsch, M., Simančík, F.: Unchain my $\mathcal{EL}$ reasoner. In: Proc. 23rd Int. Workshop on Description Logics (DL'10). CEUR Workshop Proceedings, vol. 745. CEUR-WS.org (2011)
5. Lawley, M.J., Bousquet, C.: Fast classification in Protégé: Snorocket as an OWL 2 EL reasoner. In: Proc. 6th Australasian Ontology Workshop (IAOA'10). pp. 45–49. Australian Computer Society Inc. (2010)
6. Mendez, J., Ecke, A., Turhan, A.Y.: Implementing completion-based inferences for the $\mathcal{EL}$-family. In: Proc. 23rd Int. Workshop on Description Logics (DL'10). CEUR Workshop Proceedings, vol. 745. CEUR-WS.org (2011)
7. Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C. (eds.): OWL 2 Web Ontology Language: Profiles. W3C Recommendation (27 October 2009), available at http://www.w3.org/TR/owl2-profiles/