First Results on How to Certify Subsumptions Computed by the $\mathcal{EL}$ Reasoner Elk Using the Logical Framework with Side Conditions*

Franz Baader[1], Patrick Koopmann[1], and Cesare Tinelli[2]

[1] Theoretical Computer Science, TU Dresden, Germany
firstname.lastname@tu-dresden.de
[2] Department of Computer Science, The University of Iowa, USA
cesare-tinelli@uiowa.edu

Abstract. The generation of proof certificates and the use of proof checkers is nowadays standard in first-order automated theorem proving and related areas. They have, to the best of our knowledge, not yet been employed in Description Logics, where the focus was on detecting and repairing errors in the ontology, rather than on catching erroneous consequences created by an incorrect reasoner. This paper reports on first steps towards remedying this deficit for subsumptions computed by the DL reasoner Elk. We use an existing tool for generating proofs of consequences from Elk, and transform these proofs into a format that is accepted as certificates by our proof checker. The checker is obtained as an instance of a generic certification tool based on the Logical Framework with Side Conditions (LFSC), by formalizing the inference rules of Elk in LFSC. We report on the results of applying this approach to the classification of a large number of real-world OWL 2 EL ontologies.

1 Introduction

The purpose of this paper is to show that tools developed in first-order automated theorem proving (ATP) and satisfiability modulo theories (SMT) for certifying reasoning results can in principle also be employed in Description Logic (DL) to increase the trust in reasoning results (such as the subsumption hierarchy) computed by DL reasoners.

Highly-optimized automated reasoning tools are complex software systems, and thus may produce erroneous results due to programming errors, even if soundness and completeness of the underlying calculus have been proved in detail. If the results of the reasoning process are used in safety-critical situations (e.g., when verifying software), then it is important that one can trust these results. Since it is currently not possible to verify a large and sophisticated software system like an automated theorem prover, the solution to this problem is

* Partly funded by the DFG grant 389793660 as part of TRR 248.

that the prover outputs a proof certificate for the result, which can then easily be checked using a proof checker. In contrast to the provers, proof checkers are rather simple pieces of software, and thus are easier to trust or verify. For this reason, the generation of proof certificates is now common in (general-purpose) ATP [33,29] and more specialized related areas such as SMT [31,8]. For example, in most divisions of the ATP system competition CASC [32], it is required that the participating systems output proofs for theorems and (finite) counter-models for non-theorems, though this output is not checked automatically. Proofs are used in particular when ATP and SMT tools are integrated in other reasoning tools such as skeptical proof assistants [2,9] or certifying software verifiers [22].

Since most DLs [6] are decidable fragments of first-order logic, dedicated decision procedures for specific DLs are usually more efficient for reasoning on DL knowledge bases than general-purpose theorem provers. Though DL reasoning is simpler than ATP, efficient DL reasoners[1] employ sophisticated optimizations and data structures, and are thus again complex software systems that may contain programming errors. In addition, while the correctness of the basic calculi (such as tableaux or consequence-based calculi) may have been proved in detail, this is not always the case for optimized variants. Nevertheless, the DL community has mainly concentrated on explaining [15] and repairing [16,7] errors in the input (i.e., the ontology that is classified), rather than on catching errors created by the reasoning process. In the 2015 OWL Reasoner Evaluation, the results produced by the reasoners were "validated by comparison between competitors using a majority vote/random tie-breaking fallback strategy" [26], which in some cases unfairly penalized a correct reasoner [20].

This paper reports on first steps towards remedying this deficiency, where we mainly tried to employ existing software rather than implementing new one. As reasoning task we consider classification for OWL 2 EL ontologies,[2] i.e., the computation of the subsumption hierarchy between the concepts defined in the given ontology. Our aim is to certify subsumption relationships that have been computed by the consequence-based DL reasoner Elk [19]. For this purpose, we use the inference tracing approach of [18] extending Elk to extract proofs of computed subsumptions, and then transform these proofs into a format that is accepted as certificates by our proof checker. This checker is obtained by instantiating a generic checker based on the Logical Framework with Side Conditions (LFSC) [31] with an encoding in LFSC of a sufficiently large subset of Elk's inference rules. LFSC is a meta-framework based on the established Edinburgh LF framework [14], which combines ease of presenting proof systems with a high-performance checker for proofs in the represented systems.

We have evaluated our approach on a large number of real-world OWL 2 EL ontologies, obtaining promising results. Our experiments show that the certificates are usually of manageable size and can be checked in reasonable time,

[1] See, e.g., [26] for a list of the reasoners that participated in the OWL Reasoner Evaluation (ORE) in 2015.

[2] OWL 2 EL is a profile of the standard Web Ontology Language OWL 2 https://www.w3.org/TR/owl2-profiles/.

though creating them may take a relatively long time. The latter problem could be mitigated by developing reasoners that directly generate certificates, rather than extracting them from outputs of existing tools not built for this purpose.

2 Certificate Generation and Proof Checking

Certificate generation in ATP and SMT takes various forms depending on the query issued to the prover and its result. When the query consists in showing that a goal formula φ is entailed by a set of hypotheses Γ and the prover succeeds in proving that, the generated certificate typically consists in a term that encodes a proof of φ from Γ or, equivalently, a proof of the unsatisfiability of $\Gamma \cup \{\neg\varphi\}$. In the DL context, φ could be a subsumption statement $C \sqsubseteq D$ and Γ an ontology. If the subsumption holds, then consequence-based calculi provide us with a proof of φ from Γ whereas tableau-based calculi yield a proof of the unsatisfiability of Γ with an a ABox of the form $\{C(a), \neg D(a)\}$, which describes a counterexample to the subsumption statement.

The encoding of proofs varies depending on the proof system in which the proof is expressed and the granularity of proof terms. Some provers, such as Vampire [28] and Z3 [23], generate proof terms that are in fact proof *sketches*: an external checker needs proof search capabilities in order to reconstruct a full proof from the sketch. Others, such as veriT [11], provide fine-grained proof terms that can be checked with no proof search but require the checker to provide native support for certain data structures (such as sets, clauses, or sequents) used in the proof term. Finally, other provers, such as CVC4 [13], provide fine-grained proofs as terms in a *logical framework* expressive enough to formalize also the proof system the proof is based on (such as LF [14], ELF [27] or the $\lambda\Pi$ calculus [3]). The latter require a proof checker that can take as input both the proof term to be checked and its proof system. Logical frameworks are typically based on a dependently-typed higher-order logic. This provides not only representational power but also the ability to uniformly recast proof checking as type checking. A prover's proof system is modeled as a type system T, and a proof term represents a correct proof exactly when it is well typed in T.

The last approach to proof checking provides the highest level of flexibility because the same proof checker can be used for very different provers and proof systems as long as those systems are representable in the underlying logical framework. The approach also provides a high level of trust in principle, for two reasons. First, a generic checker that has been used successfully with many proof systems is arguably more trustworthy than one custom-made for a specific proof system, unless the latter is very simple. Second, because the proof system is an input to the proof checker, it is expressed as set of proof rules whose soundness can be proved separately with a proof assistant, such as Coq [34] or Lean [24], also based on a dependently-typed higher-order logic. This effectively removes the rules from the *trusted core*, which then reduces to just the proof checker.

$$\text{(Kinds)}\quad \kappa ::= \textbf{type} \mid \Pi x{:}\tau.\,\kappa \qquad \text{(Types)}\quad \tau ::= \mathsf{int} \mid k \mid \tau\ t \mid \Pi x{:}\tau_1[\{p\ t\}].\,\tau_2$$
$$\text{(Terms)}\quad t ::= x \mid c \mid t{:}\tau \mid \lambda x[{:}\tau].\,t \mid t_1\ t_2$$

Fig. 1: **LFSC Abstract Syntax.** Letter c denotes (possibly higher-order) term constants, k type constants, x term variables, p side condition programs. Optional syntax components are in square brackets.

The Logical Framework with Side Conditions

For this work, we have used the LFSC [31], a logical framework that extends LF, the Edinburgh Logical Framework [14], with a bare-bones functional programming language to express procedural *side conditions* for proof rules. Intuitively, the extension to side conditions allows one to define proof systems more compactly, with proof rules that delegate low-level checks on the rule's premises (such as, for instance, that a given term occurs in a given premise) to a side program. We refer the interested reader to Stump et al. [31] for more details on the use of side conditions and focus on the main language instead.

A slightly *simplified* version of LFSC's abstract syntax is provided in Figure 1. It is an extension of LF's $\lambda\Pi$ calculus, itself an extension of the simply typed λ-calculus. The $\lambda\Pi$ calculus has three levels of semantic entities, all denoted by terms: *values*; *types*, understood as collections of values; and *kinds*, families of types. The constant **type** denotes the kind of types. Types and kinds can be *dependent* on (i.e., indexed by) values. Syntactically, if $\tau_2[x]$ is a type term whose set of free term variables is $\{x\}$ and τ_1 is a type term with no free variables, the expression $\Pi x{:}\tau_1.\tau_2[x]$ denotes in the calculus the (dependent) type of functions that return a value of type $\tau_2[v]$ for each value v of type τ_1 for input x. When τ_2 has no free variables, the type $\Pi x{:}\tau_1.\tau_2$ is just the type $\tau_1 \to \tau_2$ of simply typed λ-calculus,[3] and we will use the latter notation for it, treating $\to$ as right associative. The same sort of parametrization applies to kinds as well, allowing one for instance to define the type of vectors of size n with a type constant vec of kind $\Pi n{:}\mathsf{int}.\,\textbf{type}$ where int is the predefined type of mathematical integers. LFSC adds to the $\lambda\Pi$ calculus the possibility of imposing restrictions on the parameters of a depended type. These restrictions are expressed operationally by side conditions of the form $\{p\ t\}$ where p is a program in the side condition language and t is an LF term. The restriction, enforced at type checking time, is that p does not fail and its result is equal to (more precisely, matches) t.

The LFSC Checker

The LFSC checker developed at the University of Iowa is a small, high performance type checker for LFSC written in C++.[4] It takes as input one or more

[3] The type of unary functions with inputs of type τ_1 and outputs of type τ_2.
[4] The checker is available in source form at https://github.com/CVC4/LFSC.

signature files containing an LFSC encoding of a proof system with side conditions, and a file containing the certificate, the proof term to be checked, and reports whether the term is a correct proof or not. For any of the LFSC checker's applications, the trusted core consists of the signature(s) used and the checker itself which has, however, a rather small code base. Although the checker was developed originally to check SMT proofs [31], and is used mainly for that purpose,[5] thanks to its generality it has also been used for other kinds of reasoners, such as SAT solvers [25], model checkers [22] and, in our case, DL reasoners. In fact, we argue that the LFSC checker (or similar logical framework-based checkers such as Dedukti [3]) is particularly well-suited for Description Logics given the large number of such logics and proof systems for them. In this work, we chose the proof system adopted by the high-performance reasoner Elk for the logic $\mathcal{EL}$ [4] mainly because Elk produces proof traces for its subsumption checks, which can be converted to full LFSC proofs. The next section explains how we did that and how we formalized ELK's proof system in LFSC.

3 Certifying OWL EL Classification Results

We focus on the subset of OWL 2 EL corresponding to $\mathcal{EL}^+_\bot$ [4,6] since it is supported by Elk and is expressive enough to provide us with a large number of real-world ontologies for our experiments. Starting with mutually disjoint sets $\mathsf{N_C}$ and $\mathsf{N_R}$ of *concept* and *role names*, respectively, $\mathcal{EL}^+_\bot$ *concepts* are constructed from concept names using the constructors top concept ($\top$), bottom concept ($\bot$), conjunction ($C \sqcap D$), and existential restriction ($\exists r.C$). $\mathcal{EL}^+_\bot$ ontologies are then finite sets of axioms of the form $C \sqsubseteq D$ for concepts C, D (concept inclusion, CI) and $r_1 \circ \ldots \circ r_n \sqsubseteq r$ for role names $r_1, \ldots, r_n, r$ (role inclusion, RI). The semantics of $\mathcal{EL}^+_\bot$ concepts and ontologies, which defines how interpretations $\mathcal{I}$ assign sets $C^\mathcal{I}$ to concepts C and under what conditions an interpretation is a model of an ontology, is formalized in the usual way (see [4,6] for details).

One of the most important inference problems is *subsumption*: given $\mathcal{EL}^+_\bot$ concepts C, D and an $\mathcal{EL}^+_\bot$ ontology $\mathcal{O}$, we say that C is *subsumed* by D w.r.t. $\mathcal{O}$ (written $C \sqsubseteq_\mathcal{O} D$) if $C^\mathcal{I} \subseteq D^\mathcal{I}$ holds for all models $\mathcal{I}$ of $\mathcal{O}$. The subsumption problem in $\mathcal{EL}^+_\bot$ is known to be decidable in polynomial time [4]. DL systems offer *classification* as basic inference service, i.e., when reading in an ontology $\mathcal{O}$, they usually compute all subsumption relationships between the concept names occurring in $\mathcal{O}$, as well as $\top$ and $\bot$. Some DL systems actually compute (and store) not the full set of these subsumptions, but their *transitive reduct*, by leaving out those relationships that can be obtained by transitivity from others. Whereas systems that use tableaux-based subsumption reasoning realize classification by repeated calls of a basic subsumption algorithm [5], consequence-based approaches [4,17] classify the whole ontology in one go.

[5] The SMT solvers CVC3 and CVC4 produce LFSC proofs.

$$\mathsf{R}_0\frac{}{C \sqsubseteq C} \qquad \mathsf{R}_\top\frac{}{C \sqsubseteq \top} \qquad \mathsf{R}^-_\sqcap\frac{C \sqsubseteq D \sqcap E}{C \sqsubseteq D \quad C \sqsubseteq E} \qquad \mathsf{R}^+_\sqcap\frac{C \sqsubseteq D \quad C \sqsubseteq E}{C \sqsubseteq D \sqcap E}$$

$$\mathsf{R}_\exists\frac{C \sqsubseteq \exists r.D \quad D \sqsubseteq E}{C \sqsubseteq \exists r.E} \qquad \mathsf{R}_\bot\frac{C \sqsubseteq \exists r.D \quad D \sqsubseteq \bot}{C \sqsubseteq \bot} \qquad \mathsf{R}_\sqsubseteq\frac{C \sqsubseteq D}{C \sqsubseteq E} : D \sqsubseteq E \in \mathcal{O}$$

$$\mathsf{R}_\circ\frac{C_0 \sqsubseteq \exists r_1.C_1 \;\; C_1 \sqsubseteq \exists r_2.C_2 \;\; \ldots \;\; C_{n-1} \sqsubseteq \exists r_n.C_n}{C_0 \sqsubseteq \exists r.C_n} : r_1 \circ \ldots \circ r_n \sqsubseteq r \in \mathcal{O}$$

Fig. 2: Basic calculus used by ELK.

$$\mathsf{conc}, \mathsf{role}, \mathsf{axiom}, \mathsf{ont} : \textbf{type} \qquad \mathsf{h} : \mathsf{axiom} \to \textbf{type}$$

$$\_ \sqsubseteq \_ : \mathsf{conc} \to \mathsf{conc} \to \mathsf{axiom} \qquad \emptyset : \mathsf{ont} \qquad \_\,;\,\_ : \mathsf{axiom} \to \mathsf{ont} \to \mathsf{ont}$$

$$\bot, \top : \mathsf{conc} \qquad \_ \sqcap \_ : \mathsf{conc} \to \mathsf{conc} \to \mathsf{conc} \qquad \exists : \mathsf{role} \to \mathsf{conc} \to \mathsf{conc}$$

$$\mathsf{R}_0 : \begin{array}{l}\Pi c{:}\mathsf{conc}.\\ \mathsf{h}(c \sqsubseteq c)\end{array} \qquad \mathsf{R}^-_{\sqcap 1} : \begin{array}{l}\Pi c, d, e{:}\mathsf{conc}.\\ \mathsf{h}(c \sqsubseteq d \sqcap e) \to \mathsf{h}(c \sqsubseteq d)\end{array} \qquad \mathsf{R}^-_{\sqcap 2} : \begin{array}{l}\Pi c, d, e{:}\mathsf{conc}.\\ \mathsf{h}(c \sqsubseteq d \sqcap e) \to \mathsf{h}(c \sqsubseteq e)\end{array}$$

$$\mathsf{R}_\top : \begin{array}{l}\Pi c{:}\mathsf{conc}.\\ \mathsf{h}(c \sqsubseteq \top)\end{array} \qquad \mathsf{R}^+_\sqcap : \begin{array}{l}\Pi c, d, e{:}\mathsf{conc}.\\ \mathsf{h}(c \sqsubseteq d) \to \mathsf{h}(c \sqsubseteq e) \to \mathsf{h}(c \sqsubseteq d \sqcap e)\end{array} \qquad \mathsf{R}_\exists : \begin{array}{l}\Pi c, d, e{:}\mathsf{conc}.\ \Pi r{:}\mathsf{role}.\\ \mathsf{h}(c \sqsubseteq \exists\ r\ d) \to \mathsf{h}(d \sqsubseteq e) \to \mathsf{h}(c \sqsubseteq \exists\ r\ e)\end{array}$$

$$\mathsf{R}_\bot : \begin{array}{l}\Pi c, d{:}\mathsf{conc}.\ \Pi r{:}\mathsf{role}.\\ \mathsf{h}(c \sqsubseteq \exists\ r\ d) \to \mathsf{h}(d \sqsubseteq \bot) \to \mathsf{h}(c \sqsubseteq \bot)\end{array} \qquad \mathsf{R}_\sqsubseteq : \begin{array}{l}\Pi c, d, e{:}\mathsf{conc}.\ \Pi o{:}\mathsf{ont}\ \{(\mathsf{in}\ \ d \sqsubseteq e\ \ o)\ \ \mathsf{tt}\}.\\ \mathsf{h}(c \sqsubseteq d) \to \mathsf{h}(c \sqsubseteq e)\end{array}$$

Fig. 3: Partial LFSC encoding of the ELK calculus.

How Elk Proves Subsumption

For $\mathcal{EL}^+_\bot$, ELK follows a consequence-based approach for classification [19], which basically starts with the CIs in $\mathcal{O}$, and then uses classification rules (see Figure 2 for example rules) to add derived subsumptions. This saturation process terminates after a polynomial number of rule applications, and a subsumption between concept names in $\mathcal{O}$ follows from $\mathcal{O}$ iff it is contained in the saturated set computed by ELK. Note that, for efficiency reasons, in some of the rules the axioms of the ontology $\mathcal{O}$ are distinguished from the derived subsumptions by being used as side conditions rather than as premises.

The OWL 2 EL standard defines a range of axioms that can be seen as syntactic sugar for $\mathcal{EL}^+_\bot$-axioms. In addition to the rules shown here, ELK uses rules that convert those axioms into CIs or RIs, and some other rules we do not discuss for space constraints. To allow the certification of ELK proofs, we designed and implemented an LFSC signature specifying datatypes and rules corresponding to the language constructs and rules used in ELK. In the source code of ELK, we found 50 classes implementing the interface `ElkInference` for rule applications. We restricted ourselves to the inferences on (syntactic variations of) $\mathcal{EL}^+_\bot$-axiom, and identified the rules actually used in the corpora discussed in the next section. This resulted in a set of 18 rules which we then implemented in LFSC.

$$\{P \sqsubseteq R \sqcap S,\ P \sqsubseteq \exists r.T,\ T \sqsubseteq U\} \subseteq \mathcal{O}$$

$$\dfrac{\dfrac{\dfrac{}{P \sqsubseteq P}\,R_0}{P \sqsubseteq \exists r.T}\,R_{\sqsubseteq} \quad \dfrac{\dfrac{}{T \sqsubseteq T}\,R_0}{T \sqsubseteq U}\,R_{\sqsubseteq}}{P \sqsubseteq \exists r.U}\,R_{\exists} \quad \dfrac{\dfrac{}{P \sqsubseteq P}\,R_0}{P \sqsubseteq R \sqcap S}\,R_{\sqsubseteq}}{P \sqsubseteq \exists r.U \sqcap R}\,R^{-}_{\sqcap 1}\,R^{+}_{\sqcap}$$

(a) ELK proof

```
P, R, S, T, U : conc      r : role
o = P ⊑ R ⊓ S ; P ⊑ ∃ r T ; T ⊑ U ; ∅
p = R0 (P ⊑ P)
check (R+⊓ - - -
    (R∃ - - - -
      (R⊑ - - - o p)
      (R⊑ - - - o (R0 (T ⊑ T))))
    (R−⊓1 - - - (R⊑ - - - o p))
) : h(P ⊑ ∃ r U ⊓ R)
```

(b) LFSC proof

Fig. 4: From ELK to LFSC proofs.

Encoding Elk Proofs in LFSC

For illustration purposes, Figure 3 contains a fragment of the LFSC signature, in *abstract* syntax, showing how the first seven ELK rules from Figure 2 could be encoded in LFSC.[6] The first three rows declare a number of types (with conc for concepts, ont for ontologies, and so on) and constants corresponding to the symbols of $\mathcal{EL}^+_\bot$. For increased readability, for constants we use the same symbols as the corresponding operator in $\mathcal{EL}^+_\bot$. The subset of ontology axioms used in the proof are encoded as a sequence of axiom constructed with the operators $\emptyset$ and ;. The type constant h is used to construct *proof judgments*, statements of the form $\mathsf{h}(a)$ expressing that axiom a is provable.

A proof rule is represented by a dependently-typed constant whose type directly encodes the rule's premises and conclusion. We name each such constant as its corresponding rule in Figure 2, except for rules $\mathsf{R}^-_{\sqcap 1}$ and $\mathsf{R}^-_{\sqcap 2}$ which correspond respectively to the first and the second conclusion of rule $\mathsf{R}^-_{\sqcap}$ from that figure. For brevity, we use notation like $\Pi x, y{:}\tau_1.\,\tau_2$ as a shorthand for $\Pi x{:}\tau_1.\,\Pi y{:}\tau_1.\,\tau_2$. The type of each rule is parametrized by term variables that correspond to the schema variables in the ELK rule. For instance, $\mathsf{R}^+_{\sqcap}$, which has type

$$\Pi c{:}\mathsf{conc}.\Pi\, d{:}\mathsf{conc}.\Pi\, e{:}\mathsf{conc}.\,\mathsf{h}(c \sqsubseteq d) \to \mathsf{h}(c \sqsubseteq e) \to \mathsf{h}(c \sqsubseteq d \sqcap e),$$

is parametrized by variables c, d, and e of type conc which correspond, respectively, to the schema variables C, D, and E of the corresponding ELK rule. The return type $\mathsf{h}(c \sqsubseteq d \sqcap e)$ of $\mathsf{R}^+_{\sqcap}$ encodes the conclusion $C \sqsubseteq D \sqcap E$ of the ELK rule whereas the argument types $\mathsf{h}(c \sqsubseteq d)$ and $\mathsf{h}(c \sqsubseteq e)$ encode the two premises $C \sqsubseteq D$ and $C \sqsubseteq E$. The other rules are similar. Note that the type of $\mathsf{R}_{\sqsubseteq}$ encodes the side condition of the corresponding ELK rule by means of an LFSC side condition that checks whether the axiom $c \sqsubseteq d$ occurs in the input ontology o. This is done by the side condition program in (whose definition is not shown), which scans o looking for $c \sqsubseteq d$ and returns tt if, and only if, it finds it.

[6] The actual LFSC signature for the ELK calculus, including the side condition code, is provided in full in the appendix in the Lisp-like concrete syntax of LFSC.

Figure 4 shows side-to-side an ELK proof tree deriving the subsumption $P \sqsubseteq \exists r.U \sqcap R$ from an ontology containing the axioms $P \sqsubseteq R \sqcap S$, $P \sqsubseteq \exists r.T$, and $T \sqsubseteq U$, and a possible encoding of this proof and relevant subontology in LFSC. The encoding starts by declaring constants for the various concepts and roles used in the proof. It then defines variable o as an ontology (i.e., a value of type ont) consisting of all the relevant axioms. The proof is encoded as the proof term

$$\begin{array}{l}(\mathsf{R}^{+}_{\sqcap} \_ \_ \_ (\mathsf{R}_{\exists} \_ \_ \_ \_ (\mathsf{R}_{\sqsubseteq} \_ \_ \_ o\ (\mathsf{R}_0\ (\mathrm{P} \sqsubseteq \mathrm{P}))) (\mathsf{R}_{\sqsubseteq} \_ \_ \_ o\ (\mathsf{R}_0\ (\mathrm{T} \sqsubseteq \mathrm{T})))) \\ \qquad (\mathsf{R}^{-}_{\sqcap 1} \_ \_ \_ (\mathsf{R}_{\sqsubseteq} \_ \_ \_ o\ (\mathsf{R}_0\ (\mathrm{P} \sqsubseteq \mathrm{P})))))\end{array}$$

with the optimization that the two occurrences of subterm $(\mathsf{R}_0\ (\mathrm{P} \sqsubseteq \mathrm{P}))$ are factored out by the defined constant p. In each rule application, underscores are used for arguments whose value does not need to be specified as it can be generated by type inference from the remaining arguments. The term directly represents the proof tree of the ELK proof. Intuitively, for the inner rule applications, the result of the rule (its conclusion) is given as input to the surrounding rule application. The term is a correct proof of $P \sqsubseteq \exists r.U \sqcap R$ exactly if it is well typed and has type $\mathsf{h}(P \sqsubseteq \exists\ r\ U \sqcap R)$. Using the type ascription operator :, the check command directs the LFSC checker to verify that.

Extracting Proofs Using ELK's Explanation Servive

To generate LFSC proofs that can serve as certificates, we used the explanation service implemented in ELK 0.5 [18]. Ideally, these certificates would be created during classification and returned together with the classification result to the user. As a first step towards this, we used the proof extraction methods that are provided by ELK 0.5 [18] and can be direcly used from within Java. While Kazakov et al. [18] describe algorithms for extracting proofs that could serve as certificates, the implementation available in ELK 0.5 is tailored towards the explanation service within the OWL ontology editor Protégé,[7] which presents some limitations for our use case. The aim of the explanation service in Protégé is to provide users with detailed explanations of the different inference steps that could lead to a particular concept inclusion. If an axiom can be inferred in different ways using ELK, the front-end shows all these possibilities. Users can select the inference they are interested in and ask for inferences of each of the premises, thus exploring different proofs step-wise for the concept inclusion of interest. In a similar way, we use this service to reconstruct a proof automatically: for a given axiom derived by ELK, we can ask for the possible rule applications that have this axiom as conclusion. To construct the whole proof, we select one such rule application, and then iteratively continue on the premises. A naive implementation of this approach would lead to termination problems, since the same axiom may occur twice in the resulting structure.

This non-termination issue is solved in Algorithm 1 by keeping track, in the variable inferences, of the set of inferences (i.e., rule applications including

[7] https://protege.stanford.edu/

Algorithm 1 Algorithm used for generating proofs.

```
1:  knownProofs ← new Map[Axiom, Proof]
2:
3:  procedure GENERATEPROOF(α : Axiom, inferences : Set[Inference])
4:      if α ∈ knownProofs.keys then
5:          return knownProofs.get(α)
6:      else
7:          for all inf ∈ (getInferencesFor(α) \ inferences) do
8:              proofs ← {generateProof(β, inferences ∪ {inf}) | β ∈ inf.premises}
9:              if fail ∉ proofs then
10:                 proof ← (α, inf, proofs)
11:                 knownProofs.put(α ↦ proof)
12:                 return proof
13:             end if
14:         end for
15:         return fail
16:     end if
17: end procedure
```

premises and conclusion) for the current branch of the proof being constructed. To generate a proof of an axiom α, we start with the call generateProof$(\alpha, \emptyset)$ with an empty set of inferences. Then we ask ELK via getInferencesFor(α) for inference steps, but consider only those inferences that are not already in inferences. For each inference inf, we call the procedure recursively on the premises involved, adding the current inference to the set inferences (Line 8). If the proof construction did not fail for any premise, we have constructed the proof and can return it (Line 12). To further speed up the computation, we use a global cache to store previously constructed proofs or sub-proofs (Line 1, 4, 5 and 11).

4 Evaluation

We used our approach to generate certificates for $\mathcal{EL}^+_\bot$ ontologies and verified them using LFSC. Our experiments, including classification, certificate generation, and certification, were performed on an Intel(R) Core(TM) i5-4590 CPU with 4 cores at 3.30GHz and 32 GB RAM. The operating system was Debian GNU/Linux 9. The code is available online.[8]

We generated classification certificates for two corpora of $\mathcal{EL}^+_\bot$ ontologies: the OWL EL classification track of the OWL Reasoner Evaluation 2015 (ORE 2015) [26], and the Manchester OWL Corpus (MOWLCorp) [21]. From the corpora, we removed all ontologies that (1) contained axioms not corresponding to (syntactical variants of) $\mathcal{EL}^+_\bot$-axioms; or (2) were inconsistent. These two cases only applied to ontologies in MOWLCorp. For each ontology, we used ELK to generate the transitive reduct of the classification result, that is, we only included strict subsumptions that could not be derived via transitivity from

[8] https://lat.inf.tu-dresden.de/dl2020-certifying-classification-results

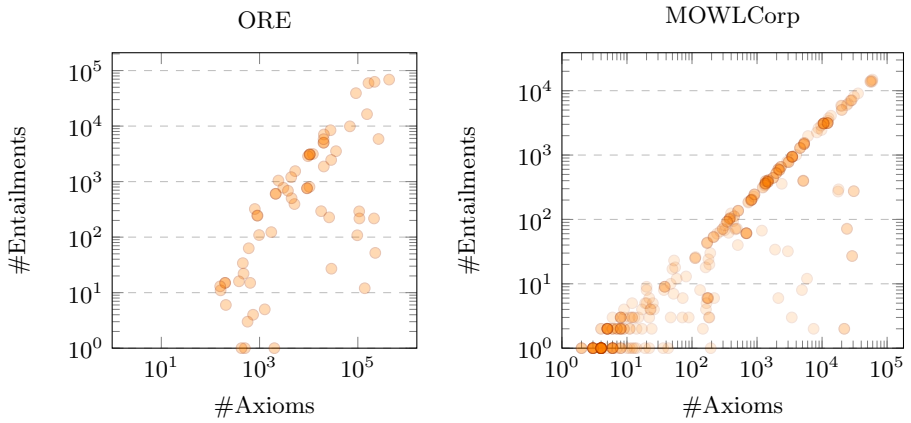

Fig. 5: Number of axioms and entailments in the ontologies of the two corpora.

others, while also taking care that equivalence classes of concepts stayed intact. Specifically, for each such equivalence class, we would pick an arbitrary cyclic chain of subsumption relations to be included. Furthermore, we excluded concept inclusions/equivalences that were explicitly stated in the ontology. In the following, when talking of the classification result, we always refer to the set of inferred subsumption relationships obtained this way, that is, the transitive reduct without explicitly stated concept inclusions and equivalences. We also removed ontologies for which the transitive closure of the stated subsumptions produced the whole hierarchy. After these removals, the ORE corpus contains 62 ontologies and the MOWLCorp corpus contains 310 ontologies. Figure 5 shows the number of axioms and the number of entailments to be verified.

We computed certificates for each classification result and verified them with the LFSC checker. For one of the ontologies from MOwlCorp, we terminated the certificate generation after 16 hours, while for all ontologies in ORE certificate was generated. In Figure 6, we show for each of the ontologies the time taken for classification, certificate generation, and verification by LFSC, where the x-axis shows the number of entailments that was certified. One can see that certificate generation took significantly longer than the classification task itself, while certificate verification took significantly less time in almost all cases. The long time for certificate generation is to due the backtracking approach used to reconstruct the proofs with Elk. We expect this to incur a much smaller overhead if certificates were generated directly during reasoning. It is therefore more insightful to look at the sizes of the generated certificates. In Figure 7, we plot those sizes against the number of entailments in each ontology. Most certificates had a size between 1 KB and 1 MB, with the largest certificate being 1.43 GB. Note that these are certificates for the whole subsumption hierarchy, and not for single subsumptions. Certificates for single subsumptions are quite small. Despite their large sizes, the certificates were verified by the LFSC checker in the order of milliseconds. Since we generated proofs for each entailment sep-

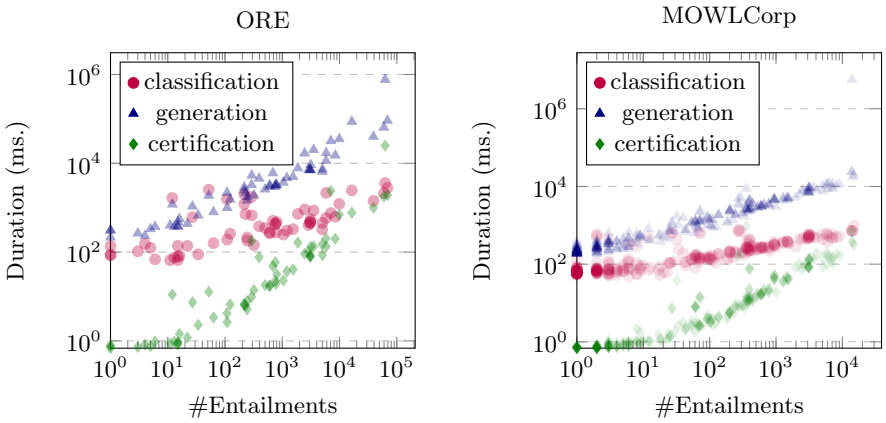

Fig. 6: Running times of classification, certificate generation, and verification.

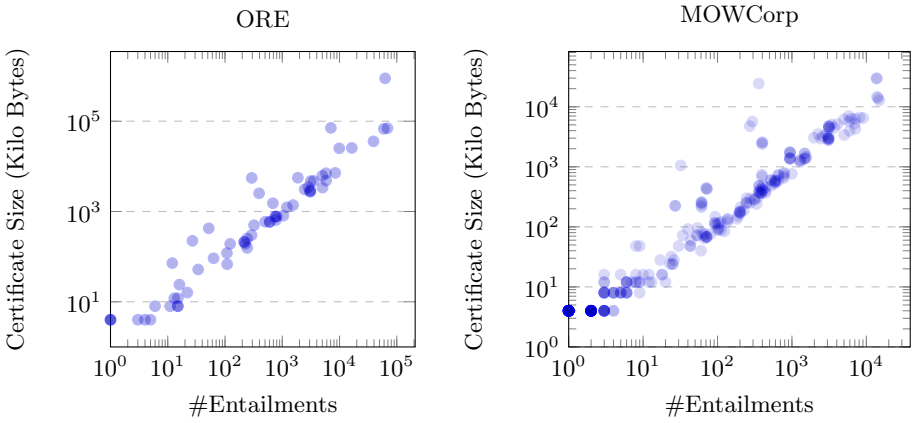

Fig. 7: File sizes of the certificates generated.

arately, our classification certificates collectively contained a lot of redundancy. By caching and sharing subproofs, the cumulative size of the certificates for all the entailments could be significantly reduced.

5 Conclusion

We have implemented a prototypical proof certification system for the reasoner Elk, utilizing the proof-generation facilities available in Elk for the generation of certificates, and employing the LFSC checker as proof-checker. Our evaluation demonstrates that, even though certificates may be quite large, they can nevertheless be verified in very short time. We exploited the explanation service of Elk to generate proof certificates a posteriori. Ideally, the generation

of certificates should take place during reasoning. This would reduce certificate generation times considerably and result in smaller certificates. Before embarking on the major task of implementing a proof-producing reasoner for $\mathcal{EL}$, we found it sensible to assess first whether proof checking of DL reasoning results based on LFSC is viable in principle.

Since our focus was on evaluating certificate checking rather than certificate generation, our algorithm for extracting proofs from Elk is fairly unsophisticated and not optimal. For instance, by building the proof starting from the relevant axioms in the ontology, rather than from the conclusion, we would avoid the need for detecting cycles. This should lead to a procedure with the same (polynomial) time complexity as Elk. The Proof Utility Library PULi[9] or the techniques by Alrabbaa et al. [1] for extracting proofs of minimal size from Elk could be of further help in implementing a more efficient certificate generation service. Nevertheless, note that the size of certificates and the time required to generate them constitutes a bottleneck only if one intends to certify all reasoning results (i.e., the whole subsumption hierarchy). The current unoptimized approach to certification could already be useful in cases where a specific subsumption result by a DL reasoner is called into question; for example, if the user doubts it, or if it differs from the result produced by another reasoner.

Certification of reasoning results is even more important for more expressive DLs than the one supported by Elk since reasoners for them are more complex and so more likely to contain errors. There are reasoners for more expressive DLs, such as Avalance [35], Konklude [30] and Sequoia [12], that use consequence-based reasoning similarly to Elk, sometimes in combination with other techniques. For such reasoners, our approach should be relatively easy to adapt. To the best of our knowledge, none of the tableau-based DL reasoners generate proofs of their computed consequences, though there was some early work on how to extract sequent proofs from a run of a tableaux algorithm [10].

Encoding complex proof systems in LFSC has the cost of trusting, or proving formally, the soundness of the encoded proof rules for each system. An alternative approach would be to design a standardised, simpler proof system for all DLs, and have reasoners translate their proofs into proofs in that system. The challenge in that case is probably more of a social than a technical nature as it requires a community-wide acceptance of a common standard.

Finally, we point out that proofs, and proof checkers, can be used only to validate the *soundness* of a classification result. In principle, the *completeness* of classification results can be certified if the reasoner provides counter-examples for non-entailed subsumption relations, which could then be validated using model checking techniques for first-order logic. Because the number of non-subsumptions is usually much higher than the number of subsumptions, the challenge in this case would be finding common certificates for a large number of non-subsumptions at a time.

[9] https://github.com/liveontologies/puli

References

1. Alrabbaa, C., Baader, F., Borgwardt, S., Koopmann, P., Kovtunova, A.: Finding small proofs for description logic entailments: Theory and practice. In: Albert, E., Kovács, L. (eds.) LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 73, pp. 32–67. EasyChair (2020), https://easychair.org/publications/paper/qgX6
2. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A modular integration of SAT/SMT solvers to Coq through proof witnesses. In: Jouannaud, J., Shao, Z. (eds.) Proceedings of the First International Conference on Certified Programs and Proofs, CPP 2011. Lecture Notes in Computer Science, vol. 7086, pp. 135–150. Springer (2011). https://doi.org/10.1007/978-3-642-25379-9_12
3. Assaf, A., Burel, G., Cauderlier, R., Delahaye, D., Dowek, G., Dubois, C., Gilbert, F., Halmagrand, P., Hermant, O., Saillard, R.: Expressing theories in the $\lambda\Pi$-calculus modulo theory and in the Dedukti system. In: Ghilezan, S., Geuvers, H., Ivetic, J. (eds.) Proceedings of the 22nd International Conference on Types for Proofs and Programs, TYPES 2016. vol. 97. Novi SAd, Serbia (2016)
4. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: Kaelbling, L.P., Saffiotti, A. (eds.) Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005). pp. 364–369. Morgan Kaufmann, Los Altos, Edinburgh (UK) (2005)
5. Baader, F., Franconi, E., Hollunder, B., Nebel, B., Profitlich, H.J.: An empirical analysis of optimization techniques for terminological representation systems or: Making KRIS get a move on. Applied Artificial Intelligence. Special Issue on Knowledge Base Management **4**, 109–132 (1994)
6. Baader, F., Horrocks, I., Lutz, C., Sattler, U.: An Introduction to Description Logic. Cambridge University Press (2017)
7. Baader, F., Kriegel, F., Nuradiansyah, A., Peñaloza, R.: Making repairs in description logics more gentle. In: Thielscher, M., Toni, F., Wolter, F. (eds.) Proc. of the Sixteenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2018). pp. 319–328. AAAI Press (2018)
8. Barrett, C., de Moura, L., Fontaine, P.: Proofs in satisfiability modulo theories. In: Delahaye, D., Woltzenlogel Paleo, B. (eds.) All about Proofs, Proofs for All, Mathematical Logic and Foundations, vol. 55, pp. 23–44. College Publications, London, UK (Jan 2015)
9. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. J. Formalized Reasoning **9**(1), 101–148 (2016). https://doi.org/10.6092/issn.1972-5787/4593
10. Borgida, A., Franconi, E., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F.: Explaining $\mathcal{ALC}$ subsumption. In: Horn, W. (ed.) Proc. of the 14th Eur. Conf. on Artificial Intelligence (ECAI 2000). pp. 209–213. IOS Press (2000)
11. Bouton, T., Oliveira, D.C.B.D., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) Proceedings of the 22nd International Conference on Automated Deduction, CADE-22. Lecture Notes in Computer Science, vol. 5663, pp. 151–156. Springer (2009). https://doi.org/10.1007/978-3-642-02959-2_12
12. Cucala, D.T., Grau, B.C., Horrocks, I.: Sequoia: A consequence based reasoner for $\mathcal{SROIQ}$. In: Simkus, M., Weddell, G.E. (eds.) Proceedings of the 32nd International Workshop on Description Logics, Oslo, Norway, June 18-21, 2019. CEUR Workshop Proceedings, vol. 2373. CEUR-WS.org (2019), http://ceur-ws.org/Vol-2373/paper-27.pdf

13. Hadarean, L., Barrett, C., Reynolds, A., Tinelli, C., Deters, M.: Fine grained SMT proofs for the theory of fixed-width bit-vectors. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) Proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (Suva, Fiji). Lecture Notes in Computer Science, vol. 9450, pp. 340–355. Springer (2015). https://doi.org/10.1007/978-3-662-48899-7_24
14. Harper, R., Honsell, F., Plotkin, G.: A Framework for Defining Logics. Journal of the Association for Computing Machinery **40**(1), 143–184 (Jan 1993)
15. Horridge, M., Parsia, B., Sattler, U.: Laconic and precise justifications in OWL. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T.W., Thirunarayan, K. (eds.) 7th International Semantic Web Conference(ISWC 2008). Lecture Notes in Computer Science, vol. 5318, pp. 323–338. Springer-Verlag (2008)
16. Kalyanpur, A., Parsia, B., Sirin, E., Grau, B.C.: Repairing unsatisfiable concepts in OWL ontologies. In: Sure, Y., Domingue, J. (eds.) Proc. of the 3rd Eur. Semantic Web Conference (ESWC'06). Lecture Notes in Computer Science, vol. 4011, pp. 170–184. Springer-Verlag (2006)
17. Kazakov, Y.: Consequence-driven reasoning for Horn $\mathcal{SHIQ}$ ontologies. In: Boutilier, C. (ed.) Proc. of the 21st Int. Joint Conf. on Artificial Intelligence (IJCAI 2009). pp. 2040–2045. IJCAI/AAAI (2009)
18. Kazakov, Y., Klinov, P.: Goal-directed tracing of inferences in $\mathcal{EL}$ ontologies. In: Mika, P., Tudorache, T., Bernstein, A., Welty, C., Knoblock, C.A., Vrandecic, D., Groth, P.T., Noy, N.F., Janowicz, K., Goble, C.A. (eds.) Proc. of the 13th International Semantic Web Conference (ISWC 2014). Lecture Notes in Computer Science, vol. 8797, pp. 196–211. Springer (2014)
19. Kazakov, Y., Krötzsch, M., Simancik, F.: The incredible ELK - from polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies. J. Autom. Reasoning **53**(1), 1–61 (2014)
20. Lee, M., Matentzoglu, N., Parsia, B., Sattler, U.: A multi-reasoner, justification-based approach to reasoner correctness. In: Arenas, M., Corcho, Ó., Simperl, E., Strohmaier, M., d'Aquin, M., Srinivas, K., Groth, P.T., Dumontier, M., Heflin, J., Thirunarayan, K., Staab, S. (eds.) Proc. of the 14th International Semantic Web Conference (ISWC 2015). Lecture Notes in Computer Science, vol. 9367, pp. 393–408. Springer (2015)
21. Matentzoglu, N., Bail, S., Parsia, B.: A snapshot of the OWL Web. In: Alani, H., Kagal, L., Fokoue, A., Groth, P.T., Biemann, C., Parreira, J.X., Aroyo, L., Noy, N.F., Welty, C., Janowicz, K. (eds.) The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference, Sydney, NSW, Australia, October 21-25, 2013, Proceedings, Part I. Lecture Notes in Computer Science, vol. 8218, pp. 331–346. Springer (2013). https://doi.org/10.1007/978-3-642-41335-3_21, https://doi.org/10.1007/978-3-642-41335-3_21
22. Mebsout, A., Tinelli, C.: Proof certificates for SMT-based model checkers for infinite-state systems. In: Piskac, R., Talupur, M. (eds.) Formal Methods in Computer-Aided Design (FMCAD 2016). pp. 117–124. IEEE (2016)
23. de Moura, L.M., Bjørner, N.: Proofs and refutations, and Z3. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) Proceedings of the LPAR 2008 Workshops Knowledge Exchange: Automated Provers and Proof Assistants. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008)
24. de Moura, L.M., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) Proceedings of the 25th International Conference on Automated Deduction, CADE-25. Lecture Notes in Computer Science, vol. 9195, pp. 378–388. Springer (2015)

25. Ozdemir, A., Niemetz, A., Preiner, M., Zohar, Y., Barrett, C.W.: Drat-based bit-vector proofs in CVC4. In: Janota, M., Lynce, I. (eds.) Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing, SAT 2019. Lecture Notes in Computer Science, vol. 11628, pp. 298–305. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_21
26. Parsia, B., Matentzoglu, N., Gonçalves, R.S., Glimm, B., Steigmiller, A.: The OWL Reasoner Evaluation (ORE) 2015 competition report. J. Autom. Reasoning **59**(4), 455–482 (2017). https://doi.org/10.1007/s10817-017-9406-8
27. Pfenning, F.: Elf: A language for logic definition and verified meta-programming. In: Proceedings of the 4th IEEE Symposium on Logic in Computer Science. pp. 313–322 (1989)
28. Reger, G.: Better proof output for Vampire. In: Kovács, L., Voronkov, A. (eds.) Proceedings of the 3rd Vampire Workshop. EPiC Series in Computing, vol. 44, pp. 46–60. EasyChair (2017)
29. Reger, G., Suda, M.: Checkable proofs for first-order theorem proving. In: Reger, G., Traytel, D. (eds.) ARCADE 2017. 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements. EPiC Series in Computing, vol. 51, pp. 55–63. EasyChair (2017). https://doi.org/10.29007/s6d1
30. Steigmiller, A., Liebig, T., Glimm, B.: Konclude: System description. J. Web Semant. **27-28**, 78–85 (2014). https://doi.org/10.1016/j.websem.2014.06.003, https://doi.org/10.1016/j.websem.2014.06.003
31. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT proof checking using a logical framework. Formal Methods in System Design **42**(1), 91–118 (2013)
32. Sutcliffe, G.: The 9th IJCAR automated theorem proving system competition – CASC-J9. AI Communications **31**(6), 495–507 (2018)
33. Sutcliffe, G., Schulz, S., Claessen, K., Gelder, A.V.: Using the TPTP language for writing derivations and finite interpretations. In: Furbach, U., Shankar, N. (eds.) Proceedings of the Third International Joint Conference on Automated Reasoning, IJCAR 2006. Lecture Notes in Computer Science, vol. 4130, pp. 67–81. Springer (2006). https://doi.org/10.1007/11814771_7
34. development team, T.C.: The coq proof assistant reference manual version 8.9 (2019), https://coq.inria.fr/distrib/current/refman/
35. Vlasenko, J., Daryalal, M., Haarslev, V., Jaumard, B.: A saturation-based algebraic reasoner for $\mathcal{ELQ}$. In: Fontaine, P., Schulz, S., Urban, J. (eds.) Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning co-located with International Joint Conference on Automated Reasoning (IJCAR 2016), Coimbra, Portugal, July 2nd, 2016. CEUR Workshop Proceedings, vol. 1635, pp. 110–124. CEUR-WS.org (2016), http://ceur-ws.org/Vol-1635/paper-10.pdf

A LFSC signature for Elk's calculus

Included below is the LFSC signature used to model the fragment of the calculus of Elk used for our experiments. The source code of our implementation, CSV files with evaluation results, and information on how to repeat the experiments are provided under

https://lat.inf.tu-dresden.de/dl2020-certifying-classification-results .

For space constraints, the paper glosses over a number of lower level technical points. For instance, the handling of role chains is more complex in practice, and an exact description would have been out of the scope of the paper. The main points are the following.

- LFSC does not support rules with a variable number of premises, which is why we had to represent rule applications of this kind by a sequence of rule application with a fixed number of premises. For the $\circ$-rule, the easiest way to do this is to allow existential role restrictions to contain role chains instead of roles, as we can then build up the role chain step-wise, before matching it with the role inclusion axiom.
- For certification purposes, it is more convenient to use the needed axioms from the ontology as proof assumptions, instead of having side conditions that check their membership in the ontology. Checking that proof assumptions are from a given ontology can be achieved with the `TBox` proof rule.
- Types `rname`, `rchain`, `concept`, `axiom`, and `ontology` are used respectively for roles, role chains, concepts, axioms, and ontologies.
- An LFSC signature is defined by a series of *commands* for declaring or defining constants, for checking the well-typedness of term, and for defining side-condition programs. New type or term constants are introduced by the `declare` command. Side condition programs are defined by the `program` command.
- The concrete syntax of LFSC is s-expression-based, as in Lisp and Scheme, so all the encoded $\mathcal{EL}$ operators are prefix operators (and typically have an alphanumeric name). Comments in the signature (lines starting with `;`) relate these operators to the $\mathcal{EL}$ ones they encode.
- While we do not use TBox axioms as side conditions in the certification, side conditions are used for various technical reasons not discussed in the paper. For instance, they are used to compute the concatenation of two role chains.

The actual LFSC signature used in the experiments follows next.

```
;; LFSC encoding of ELK proof system

;; Authors: Patrick Koopmann and Cesare Tinelli
;; Date: June 2020

;--------
; Legend
;--------

; (t1 t2)          denotes function application
; (! x τ1 τ2)        "    Πx : τ1.τ2
; (\ x t)             "    λx.t
; (% x τ t)           "    λx : τ.t
; (@ x t e)           "    let x = t in e
; (: τ t)             "    t : τ

;; Note: Except for function applications, the syntax above is
;; used in LFSC terms, not in side condition programs

;----------------
; Proof Language
;---------------

;; Note: Every user-defined type τ in LFSC is effectively
;; an algebraic datatype:
;; every function of return type τ is a constructor for τ.
;;
; Role name type
(declare rname type)
;; Note: no constructors are needed for rname,
;; using variables of type rname is enough.

; Role chain type
(declare rchain type)
;; Note: rchain is used to represent concatenations of roles
;; in (flat) normal form.

; Role chain constructors
;
; empty role chain
(declare erc rchain)
; non-empty role chain constructor
(declare rc (! x rname (! y rchain rchain)))
;; Note: rchains are essentially erc-terminated lists.


; Concept type
(declare concept type)
```

```
; Concept constructors
;
; atomic concept (has the form (ac 0), (ac 1), ...)
(declare ac (! n mpz concept))
;; Note: mpz is the builtin type in LFSC of (infinite precision)
;; integers. The usual integer operators are built in.
;
; ⊤
(declare top concept)
; ⊥
(declare bot concept)
; _ ⊓ _
(declare inter (! x concept (! y concept concept)))
; ∃ _._
(declare ex (! x rchain (! y concept concept)))
;; Note: The first argument of ex is a *role chain*, not a role.
;; This is for convenience.
;
;; ¬ _
(declare neg (! c concept concept))
;; Note ¬ should not be part of EL, but interestingly,
;; some rules of ELK use it.

; Axiom type
(declare axiom type)

; Axiom constructors
;
; Concept inclusion _ ⊑ _
(declare sub (! x concept (! y concept axiom)))
; Concept equivalence _ ≡ _
(declare eq (! x concept (! y concept axiom)))
; Role (chain) inclusion _ ⊑ _
(declare rsub (! x rchain (! y rchain axiom)))
; Role (chain) equivalence _ ≡ _
(declare req (! x rchain (! y rchain axiom)))
; object property domain axioms
(declare roleDomain (! x rchain (! y concept axiom)))
; object property range axioms
(declare roleRange (! x rchain (! y concept axiom)))
; transitive property axiom
(declare transRole (! r rchain axiom))

; Ontology type
(declare ontology type)

; Ontology constructors
;
; empty ontology
```

```
(declare eon ontology)
; non-empty ontology constructor
(declare on (! x axiom (! y ontology ontology)))
;; Note: An ontology is essentially a (flat, eon-termindated)
;; list of of axioms.


;-------------------------
; Side Condition Programs
;-------------------------

;; Side condition terms can appear directly in a side condition
;; or can be abstracted in named, parametrized programs.
;; Programs are functional, monomorphic, and first-order.
;; They can be  recursive but not mutually recursive. They can
;; diverge, terminate normally (returning a value), or
;; abnormally (raising a "fail" exception).
;; Inputs and output are LF terms which need not be ground,
;; that is, they can contain logical variables. Those are not
;; to be confused with program variables, i.e., input parameters
;; and local variables.
;; Local variables are introduced by the match and let
;; constructs. Program variables are lexically scoped.
;; Program evaluation is eager, with pass-by-value semantics.

; Useful for side condition programs
(declare bool type)
; bool constructors
(declare tt bool)
(declare ff bool)


; (IsIn t1 t2) returns tt if t1 occurs as an axiom of t2;
; it returns ff otherwise.
(program isIn ((a axiom) (o ontology)) bool
  (match o
    (eon ff)
    ((on a1 o1)
      ; evaluates to tt if a1 is syntactically equal to a;
      ; and to the value of (isIn a o1) otherwise
      (ifequal a1 a tt (isIn a o1)))))

;; Note: For (IsIn t1 t2) to terminate normally t2 cannot be
;; a logical variable; it has to have the form eon or
;; (on a1 (on a2 ... (on an eon) ...)), otherwise match will fail.
;; However, t1, a1, ..., an can all be or contain logical variables.

; Concatenates two role chains (like list append)
(program concat ((rc1 rchain) (rc2 rchain)) rchain
  (match rc1
```

```
    (erc rc2)
    ((rc r rs) (rc r (concat rs rc2)))))

;; Note: For similar reason as in isIn, (concat t1 t2) fails
;; if t1 is a logical variable


; (icontains c1 c2) succeeds if c1 is syntactically equal to c2
; of it is a intersection of c2 with other concepts
(program icontains ((c concept) (d concept)) bool
  (ifequal c d tt
    (match c
      ((inter c1 c2) (match (icontains c1 d) (tt tt) (ff (icontains c2
    d))))
      (top ff)
      (bot ff)
      ((ex ch c1) ff)
      ((ac c1) ff))))


;-------------
; Proof Rules
;-------------

; Proof judgment for assumed or proved axioms.
; Technically, a type dependent on input axiom c.
(declare holds (! c axiom type))


;; The proof rules are functions whose return type has the form
;; (holds a) where a is the rule's conclusion.
;; The input parameters correspond to the rule's parameters
;; and premises, if any

; TBox
; Every axiom in o is derivable
(declare TBox (! o ontology (! a axiom ; rule parameters
;                                      ; premises (none)
  (! sc (^ (isIn a o) tt)              ; side condition
; -----------------------
  (holds a)                            ; conclusion
;
))))

;; Note: For notational convenience, side conditions are
;; introduced as fake arguments whose "type" has the form
;; (^ p r) where
;; - p is a side condition program (a term in the side-condition
;;   language) and
;; - r is a term (possibly with variables).
```

```
;; When a proof rule is applied to actual arguments,
;; the side condition succeeds iff
;;   1) running p doesn't cause a fail exception and
;;   2) the result of p matches r
;; Any unbound variables in r get bound by the matching
;; substitution.
;; In TBox, p is the call (isIn a o) and r is the ground Boolean
;; term tt.

;; Note: The term (TBox t1 t2) is well typed iff
;; - t1 has type ontology,
;; - t2 has type axiom, and
;; - (isIn t1 t2) evaluates to tt
;; Since the the formal parameter sc of TBox is fictitious,
;; no actual parameter for it is needed in the rule application
;; (TBox t1 t2).


; Bottom Subclass
(declare Bottom (! c concept  ; rule parameters
;                             ; premises (none)
; -------------------
  (holds (sub bot c))         ; conclusion     ⊥ ⊑ C
;
))

; Top Superclass
(declare Top (! c concept
;
;---------------------
    (holds (sub c top))    ;  C ⊑ ⊤
))

; Class Inclusion Tautology
(declare subRef (! c concept
;
; -----------------
  (holds (sub c c))   ; C ⊑ C
;
))

; Property Inclusion Tautology
(declare rsubRef (! r rchain
;
; ------------------
  (holds (rsub r r))   ; r ⊑ r
;
))

; Intersection Decomposition 1
```

```
(declare InterDec1 (! c concept (! d concept
;
; ---------------------------
  (holds (sub (inter c d) c))    ; C ⊓ D ⊑ C
;
)))

; Intersection Decomposition 2
(declare InterDec2 (! c concept (! d concept
;
; ----------------------------
  (holds (sub (inter c d) d))    ; C ⊓ D ⊑ D
;
)))

; Intersection Decomposition
; Useful for nested intersections
(declare InterDec (! c concept (! d concept
;
  (! sc (^ (icontains c d) tt)   ; D occurs in C
; ---------------------------
  (holds (sub c d))              ; C ⊑ D
;
))))

; Intersection Composition
(declare InterComp (! c concept (! d concept (! e concept
;
  (! p1 (holds (sub c d))         ; C ⊑ D
  (! p2 (holds (sub c e))         ; C ⊑ E
; ---------------------------
  (holds (sub c (inter d e)))     ; C ⊑ D ⊓ E
;
))))))

; Existential of Bottom
(declare ExBottom (! r rchain
;
; ---------------------------
  (holds (sub (ex r bot) bot))   ; ∃r.⊥ ⊑ ⊥
;
))

; Existential Filler Expansion
(declare ExExpand (! r rchain (! c concept (! d concept
;
  (! p (holds (sub c d))            ; C ⊑ D
; -------------------------------
  (holds (sub (ex r c) (ex r d)))    ; ∃r.C ⊑ ∃r.D
;
```

```
)))))

; Existential Property Expansion
(declare ExExpandRole (! c concept (! r1 rchain (! r2 rchain
;
  (! p (holds (rsub r1 r2))            ; r1 ⊑ r2
; --------------------------------
  (holds (sub (ex r1 c) (ex r2 c)))    ; ∃r1.C ⊑ ∃r2.C
)))))


;; Note: The Existential Composition rule of ELK cannot be
;; encoded as a single LFSC rule because it has a variable
;; number of premises. So it has by several variants in LFSC.


; Existential Composition 1
(declare ExComp1 (! c0 concept (! c1 concept (! c2 concept
                 (! r1 rchain (! r2 rchain (! r rchain
;
  (! p1 (holds (sub c0 (ex r1 c1)))    ; C0 ⊑ ∃r1.C1
  (! p2 (holds (sub c1 (ex r2 c2)))    ; C1 ⊑ ∃r2.C2
  (! sc (^ (concat r_1 r_2) r)         ; r = r1 ∘ r2
; --------------------------------
  (holds (sub c0 (ex r c2)))           ; C0 ⊑ ∃r.C2
;
))))))))))))

; Existential Composition 2
(declare ExComp2 (! c0 concept (! c1 concept
                 (! r1 rchain (! r rchain
;
  (! p1 (holds (sub c0 (ex r1 c1)))   ; C0 ⊑ ∃r1.C1
  (! p2 (holds (rsub r1 r))           ; r1 ⊑ r
; --------------------------------
  (holds (sub c0 (ex r c1)))          ; C0 ⊑ ∃r.C1
;
)))))))

; Existential Composition
(declare ExComp (! c0 concept (! c1 concept (! c2 concept
                (! r1 rchain (! r2 rchain (! r12 rchain (! r rchain
;
  (! p1 (holds (sub c0 (ex r1 c1)))    ; C0 ⊑ ∃r1.C1
  (! p2 (holds (sub c1 (ex r2 c2)))    ; C1 ⊑ ∃r2.C2
  (! p3 (holds (rsub r12 r))           ; r12 ⊑ r
  (! sc (^ (concat r1 r2) r12)         ; r12 = r1 ∘ r2
; --------------------------------
  (holds (sub c0 (ex r c2)))           ; C0 ⊑ ∃r.C2
;
```

```
))))))))))))


; Class Hierarchy
(declare ConHi (! c1 concept (! c2 concept (! c3 concept
;
  (! p1 (holds (sub c1 c2))     ; C1 ⊑ C2
  (! p2 (holds (sub c2 c3))     ; C2 ⊑ C3
; -------------------------
  (holds (sub c1 c3))           ; C1 ⊑ C3
;
))))))

; Property Hierarchy
(declare RolHi (! r1 rchain (! r2 rchain (! r3 rchain
;
  (! p1 (holds (rsub r1 r2))    ; r1 ⊑ r2
  (! p2 (holds (rsub r2 r3))    ; r2 ⊑ r3
; -------------------------
  (holds (rsub r1 r3))          ; r1 ⊑ r3
))))))


; Equivalent Classes Decomposition 1
(declare EqDec1 (! c concept (! d concept
;
  (! p (holds (eq c d))     ; C ≡ D
; --------------------
  (holds (sub c d))         ; C ⊑ D
;
))))

; Equivalent Classes Decomposition 2
(declare EqDec2 (! c concept (! d concept
;
  (! p (holds (eq c d))     ; C ≡ D
; --------------------
  (holds (sub d c))         ; D ⊑ C
;
))))

;  Classes Inclusion Cycle
(declare ConCyc (! c concept (! d concept
;
  (! p1 (holds (sub c d))     ; C ⊑ D
  (! p2 (holds (sub d c))     ; D ⊑ C
; ----------------------
  (holds (eq c d))            ; D ≡ C
;
)))))
```

```
; Proof judgment used to single out a goal.
(declare goal (! c axiom type))

; Proof judgment used for convenience.
; All goal-oriented proofs end in done.
(declare done type)

; Allows one to provide a proof goal explicitly.
(declare Proved (! g axiom
;                    from:
  (! p1 (goal g)    ; a goal g to be proven and
  (! p2 (holds g)   ; a proof of g
; ---------------
  done              ; we can conclude that we are done
))))


;-----------------------------------------------------------
; Missing Rules used by ELK but not mentioned in ELK paper
;-----------------------------------------------------------

; Property Domain Translation
(declare RoleDomain (! r rchain (! c concept
;
  (! p1 (holds (roleDomain r c))   ; The domain of r is C
; ------------------------------
  (holds (sub (ex r top) c))       ; ∃r.⊤ ⊑ C
))))

; Transitive Role
(declare TransitiveRole (! r rchain (! rr rchain
;
  (! p1 (holds (transRole r))   ; r is transitive
  (! sc (^ (concat r r) rr)     ; rr = r ∘ r
; --------------------------
  (holds (rsub rr r))           ; rr ⊑ r
)))))

; Negation Clash
(declare NegationClash (! c concept
;
; ---------------------------------
 (holds (sub (inter c (neg c)) bot))   ; C ⊓ ¬C ⊑ ⊥
))
```

B Sample LFSC certificates

Included below is a file with several proof certificates The certificates were manually generated for greater readability. A few of the certificates automatically generated with our implementation are available at https://lat.inf.tu-dresden.de/dl2020-certifying-classification-results.

```
;-------------------------------------------
; Sample proof certificates for ELK signature
;-------------------------------------------

; check proof that ⊤ ⊑ ⊤
(check
(: (holds (sub top top))
   (InterDec top top)
))

; check proof that C ⊓ D ⊑ C
(check
(% C concept
(% D concept
(: (holds (sub (inter C D) C))
   (InterDec (inter C D) C)
)))))

; check proof that C ⊓ D ⊑ D
(check
(% n1 mpz
(% n2 mpz
(@ C (ac n1)
(@ D (ac n2)
(: (holds (sub (inter C D) D))
   (InterDec (inter C D) D)
))))))


; Deductive proof of Kidney ⊑ Kidney
; Shows an axiom derivable from no assumption
(check
(% Kidney concept
(: (holds (sub Kidney Kidney))
   (subRef Kidney)
)))


; Goal oriented proof of Kidney ⊑ Kidney
; the expected goal is specified beforehand
(check
; ----------------------------------------------------
```

```
; Concepts
(% Kidney concept
; ----------------------------------------------------
; Goal
(% g (goal (sub Kidney Kidney))
; ----------------------------------------------------
; Proof of goal from no assumptions
 (: done
    (Proved _ g (subRef Kidney))
 ))))

;; Note:: _ above can be used instead of the actual
;; parameter whenever the latter can be constructed
;; from the other actual parameters by type inference.
;; In this case, the inferred actual parameter is
;; (sub Kidney Kidney)


; Proof from assumptions
; Check that
;   AntiDiuresis ⊑ ∃isFunctionOf.Kidney
; follows from:
;   - AntiDiuresis ⊑ ExcretionOfUrine
;   - ExcretionOfUrine ⊑ ∃isFunctionOf.Kidney
(check
; Roles
(% isFunctionOf rname
; Concepts
(@ AntiDiuresis (ac 1)
(@ Excretion (ac 3)
(@ ExcretionOfUrine (ac 4)
(@ Kidney (ac 5)
; Goal: AntiDiuresis ⊑ ∃isFunctionOf.Kidney
(% g (goal (sub AntiDiuresis (ex (rc isFunctionOf erc) Kidney)))
; Assumptions
; AntiDiuresis ⊑ ExcretionOfUrine
(% p1 (holds (sub AntiDiuresis ExcretionOfUrine))
; ExcretionOfUrine ⊑ ∃isFunctionOf.Kidney
(% p2 (holds (sub ExcretionOfUrine (ex (rc isFunctionOf erc) Kidney)))
; Proof of goal from assumptions
 (: done
    (Proved _ g (ConHi _ _ _ p1 p2))
 ))))))))))


; Proof from ontology
;
; Check that AntiDiuresis ⊑ ∃isFunctionOf.Kidney
; follows from an ontology containing
;   - AntiDiuresis ≡ (Excretion ⊓ ∃actsSpecificallyOn.Urine
```

```
;                  ⊓ ∃hasProcessActivity.decreasedActivityLevel)
;   - AntiDiuresis ⊑ ExcretionOfUrine
;   - ExcretionOfUrine ⊑ ∃isFunctionOf.Kidney
(check
; Roles
(% actsSpecificallyOn rname
(% hasProcessActivity rname
(% isFunctionOf rname
; Concepts
(@ AntiDiuresis (ac 1)
(@ decreasedActivityLevel (ac 2)
(@ Excretion (ac 3)
(@ ExcretionOfUrine (ac 4)
(@ Kidney (ac 5)
(@ Urine (ac 6)
; Axioms
; AntiDiuresis ≡ (Excretion ⊓ ∃actsSpecificallyOn.Urine
;              ⊓ ∃hasProcessActivity.decreasedActivityLevel)
(@ a1 (eq AntiDiuresis
          (inter Excretion
            (inter (ex (rc actsSpecificallyOn erc) Urine)
              (ex (rc hasProcessActivity erc) decreasedActivityLevel))))
; AntiDiuresis ⊑ ExcretionOfUrine
(@ a2 (sub AntiDiuresis ExcretionOfUrine)
; ExcretionOfUrine ⊑ ∃isFunctionOf.Kidney
(@ a3 (sub ExcretionOfUrine (ex (rc isFunctionOf erc) Kidney))
; Ontology
(@ o (on a1 (on a2 (on a3 eon)))
; Goal: AntiDiuresis ⊑ ∃isFunctionOf.Kidney
(% g (goal (sub AntiDiuresis (ex (rc isFunctionOf erc) Kidney)))
; Assumptions
; Proof of goal from assumptions
 (: done (Proved _ g
    (ConHi _ _ _
     (TBox o a2)
     (TBox o a3)
    )
 ))))))))))))))))))
;; Note: the applications of TBox are not really needed.
;; A proof of g from explicit assumptions a2 and a3
;; should suffice as a proof certificate.


;; Proof from assumptions (no TBox applications)
(check
; Roles
(% actsSpecificallyOn rname
(% hasProcessActivity rname
(% isFunctionOf rname
; Concepts
```

```
(@ AntiDiuresis (ac 1)
(@ decreasedActivityLevel (ac 2)
(@ Excretion (ac 3)
(@ ExcretionOfUrine (ac 4)
(@ Kidney (ac 5)
(@ Urine (ac 6)
; Goal: AntiDiuresis ⊑ ∃isFunctionOf.Kidney
(% g (goal (sub AntiDiuresis (ex (rc isFunctionOf erc) Kidney)))
; Assumptions
; AntiDiuresis ≡ (Excretion ⊓ ∃actsSpecificallyOn.Urine
;                ⊓ ∃hasProcessActivity.decreasedActivityLevel)
(% p1 (holds (eq AntiDiuresis
        (inter Excretion
          (inter (ex (rc actsSpecificallyOn erc) Urine)
            (ex (rc hasProcessActivity erc)
              decreasedActivityLevel)))))
; (Excretion ⊓ ∃actsSpecificallyOn.Urine) ≡ ExcretionOfUrine
(% p2 (holds (eq (inter Excretion
                   (ex (rc actsSpecificallyOn erc) Urine))
                 ExcretionOfUrine))
; ExcretionOfUrine ⊑ ∃isFunctionOf.Kidney
(% p3 (holds (sub ExcretionOfUrine
               (ex (rc isFunctionOf erc) Kidney)))
; Proof of goal from assumptions
 (: done
    (Proved _ g
     (ConHi _ _ _
      (ConHi _ _ _
       (InterComp _ _ _
        (ConHi _ _ _
         (EqDec1 _ _ p1)
         (InterDec1
          Excretion  ; actually inferrable, kept for readability
          (inter (ex (rc actsSpecificallyOn erc) Urine)
            (ex (rc hasProcessActivity erc)
              decreasedActivityLevel))))
        (ConHi _ _ _
         (EqDec1 _ _ p1)
         (InterDec
          (inter Excretion
            (inter (ex (rc actsSpecificallyOn erc) Urine)
              (ex (rc hasProcessActivity erc)
                decreasedActivityLevel)))
          (ex (rc actsSpecificallyOn erc) Urine))))
       (EqDec1 _ _ p2))
      p3)))
 ))))))))))))))
```