Understanding LEO-II's proofs

Nik Sultana
Cambridge University, UK
nik.sultana@cl.cam.ac.uk

Christoph Benzmüller
Free University of Berlin, Berlin, Germany
c.benzmueller@googlemail.com

Abstract

The LEO and LEO-II provers have pioneered the integration of higher-order and first-order automated theorem-proving. To date, the LEO-II system is, to our knowledge, the only automated higher-order theorem-prover which is capable of generating joint higher-order–first-order proof objects in TPTP format. This paper discusses LEO-II's proof objects. The target audience are practitioners with an interest in using LEO-II proofs within other systems.

1 Introduction

The higher-order automated theorem-prover LEO [11; 7] was originally designed as a fully-automated subsystem of the interactive proof assistant and proof planner OMEGA [29][1]. Similar in spirit to Andrews' pioneering TPS system [2], LEO was intended to solve selected subgoals fully-automatically in order to save user interaction or support a proof planner. Hence, the provision of detailed proof information has always been a necessary feature of LEO's. This also holds true for LEO-II[2], which has recently been integrated with Sigma [27] – the ontology engineering environment used in the SUMO project [26]. Moreover, LEO-II is currently being integrated with Isabelle/HOL [24] and the HETS toolset [23]. We seek to present the details of LEO-II's proof objects in order to support projects such as these, which are interested in consuming LEO-II proofs.

Understanding LEO-II's proofs can be non-trivial. This is partly because LEO-II's proof search is based on a higher-order RUE-resolution approach [6; 7], a formalism that is arguably rather poorly suited for human use. Furthermore, recent versions of LEO-II also adapt normalisation techniques, such as splitting (§4.5), which might not help in producing readable proofs. Moreover, LEO-II's proof search integrates calls to automated first-order theorem-provers. By default, LEO-II is linked with the prover E [28]. As a novel side contribution of this paper, LEO-II now supports integrated higher-order–first-order proof objects – in which TPTP proof objects delivered by E are first processed by LEO-II and subsequently merged into joint LEO-II+E proof objects. An example of such a proof is given in Appendix A.

The main goal of this paper is to expose the proof calculus of the LEO-II prover. We also outline LEO-II's proof procedure, and describe some challenges in higher-order proof-search. Our motivation is mainly practical: the paper provides information needed to understand LEO-II's proof output, in order to support the reconstruction, and verification, of those proofs. Theoretical aspects will be addressed as far as required, and relevant pointers to more detailed literature will be given.

LEO-II's proof calculus is described in §4, and its proof procedure is outlined in §5. We start with some background: first on the challenges faced when automating theorem-proving in higher-order logic (§3.1), then on some features of LEO-II's implementation (§3.2). The appendix contains an example problem in THF, and the proof produced by LEO-II in collaboration with E.

[1]LEO stood for 'Logic Engine for OMEGA'

[2]LEO-II can be downloaded from www.leoprover.org. This paper deals with the current version of LEO-II (1.3).

2 Preliminaries

2.1 Syntax

The syntactic conventions used in this article are mostly conformant with those used in the higher-order automated theorem-proving (ATP) and LEO-II literature. LEO-II relies on a formalisation of higher-order logic based on simply-typed λ-calculus. We summarise the syntax of this formalism here – a more detailed description was given by Benzmüller et al. [12].

Types (which shall be ranged-over by the metavariables $\tau, \tau_1, \ldots, \sigma, \ldots$) are freely-generated from o (the type of propositions), ι (the type of individuals), and $\tau_1 \to \tau_2$ (functions from τ_1 to τ_2). We will sometimes abbreviate $\tau_1 \to \tau_2$ to $\tau_2\tau_1$. The type o contains two elements, $\mathtt{tt}$ and $\mathtt{ff}$, denoting truth and falsity respectively.

Terms are formed from variables, constants, parameters, abstractions and applications. The metavariables $\mathbf{M}, \mathbf{N}, \ldots$ range over terms. *Variables* $X^\tau, X_1^\tau, Y^\tau, \ldots$ are denoted by identifiers which start with upper-case letters, and range over values of type τ. *Constants* denote fixed logical objects, such as $\vee^{o\to o\to o}$, $\neg^{o\to o}$, and $\Pi^{(\tau\to o)\to o}$, standing for disjunction, negation, and universal quantification respectively. *Parameters* $c^\tau, c_1^\tau, \ldots, d^\tau, \ldots$ start with lower-case letters and denote some fixed value of type τ. *Abstraction* forms terms which denote functions. The abstraction term $\lambda X^\tau.\mathbf{M}$ denotes the function mapping some value X from τ to the value denoted by the term $\mathbf{M}$. We will drop the type annotations of variables or constants when this information is redundant. *Application* is indicated by juxtaposition of terms, and it denotes the application of a function to another value.

In $\lambda X^\tau.\mathbf{M}$, variable X is said to be *bound* in $\mathbf{M}$, and the scope of the binding extends to as far to the right as possible. If a variable in a term is not bound by an abstraction, then that variable is said to be *free.* A term with no free variables is said to be *closed. Formulas* are terms of type o, and *sentences* are closed formulas. $\mathbf{A}$ and $\mathbf{B}$ are metavariables ranging over sentences. A formula is said to be *atomic* if it is a propositional (i.e. having type o) parameter, or a propositional variable, or of the (application) form $c\,\mathbf{M}$, where c is not a logical constant.

LEO-II implements a resolution proof calculus; inference involves manipulating *clauses.* In this setting, a clause is a set of formulas. These formulas are implicitly disjoint. Clauses containing only atomic formulas are said to be *proper.* $\mathbf{C}$ and $\mathbf{D}$ are metavariables ranging over clauses. Members of clauses are called *literals.* Literals are formulas shown in square brackets and labelled with a *polarity* (either $\mathtt{tt}$ or $\mathtt{ff}$). For instance, the literal $[\neg X^o]^{\mathtt{ff}}$ denotes the negation of $\neg X$.

Inference rules will be shown in the usual format. In the example below, the symbol $\vee$ stands for a meta-disjunction (i.e. between literals) since it occurs outside square brackets[3]. Clauses are built using such meta-disjunctions. The example below is a dummy inference rule which picks some literal $[\mathbf{M}]^p$ and transforms it into $[\mathbf{N}]^q$. The symbol $\mathbf{C}$ denotes the rest of the clause.

$$\frac{\mathbf{C} \vee [\mathbf{M}]^p}{\mathbf{C} \vee [\mathbf{N}]^q}\,\mathsf{rule\_name}$$

We will use $[\mathbf{M}/X^\tau]$, using postfix syntax, to denote the capture-avoiding substitution of free occurrences of X with the τ-typed term $\mathbf{M}$. Substitution may be applied to both terms and clauses.

We will occasionally use the following abbreviations for n-fold application and disjunction respectively: $\mathbf{M}\,\overline{\mathbf{N}}^n$ for $\mathbf{M}\,\mathbf{N}_1 \ldots \mathbf{N}_n$, and $\overline{[\mathbf{M}_i = \mathbf{N}_i]^p}^{i\le k}$ for $[\mathbf{M}_1 = \mathbf{N}_1]^p \vee \ldots \vee [\mathbf{M}_k = \mathbf{N}_k]^p$.

[3]The symbol $\vee$ is overloaded to stand for object-level disjunction when it occurs within square brackets.

2.2 TPTP Annotated Formulas

TPTP [31] stands for 'Thousands of Problems for Theorem Provers' but it is also much else besides. It includes a family of languages for encoding input to theorem-provers. These languages include CNF (clausal first-order logic), FOF (first-order problems in 'natural form'), TFF (typed first-order form) and THF (typed higher-order form). THF [32] is a relatively recent addition to TPTP. It is a language for encoding problems in higher-order logic. Both E and LEO-II parse TPTP languages: E parses CNF and FOF, and LEO-II parses CNF, FOF and THF.

Formulas in TPTP are annotated with additional information: a *name*, *role*, and other optional information (e.g. relating to inferences). The name of an annotated formula serves to reference that formula (e.g. in proofs). Possible formula roles include: definition, axiom, lemma, and conjecture. Additional annotation information could indicate the source of a formula, or its derivation from other formulas by inference. This last kind of annotation is used to encode proofs as chains of TPTP annotated formulas. Thus, TPTP languages are not only used to encode logic problems, but also to encode their proofs. LEO-II produces such proofs (see Appendix A.2 for an example), and the goal of this article is to define the inference rules used in the higher-order segments of LEO-II proofs.

3 Background

3.1 First-order vs higher-order automated reasoning

This section outlines some core differences between first-order and higher-order *automated* theorem-proving. It is intended to support the technical description of LEO-II's calculus in the remainder of the paper. In this paper we only concern ourselves with classical logic.

Benzmüller et al. [12] give a tour of systems of models of higher-order logic (HOL). These systems form a family of weak models for HOL, for which complete calculi can be defined. In a way, equality is 'native' in HOL – for instance, the weakest of these models validates β-equivalence. The strongest of these systems is called *Henkin semantics*, and it is the semantics under which LEO-II works.

Unlike in FOL, terms in HOL have a native equality defined on them through λ-conversion. In Henkin semantics, this relation corresponds to $\alpha\beta\eta$-conversion. In HOL, terms may be function-valued, and formulas are simply Boolean-valued terms. Term equivalence is taken to be modulo λ-conversion. Terms are represented, and $\beta\eta$-reduced, in LEO-II as graphs.

Comprehension is another strength of HOL over FOL. Comprehension is a device for defining sets through formulas. In FOL, comprehension axioms need to be explicitly stated, but these axioms are native to HOL since sets are defined *as* formulas[4]. Benzmüller and Kerber [9] identify comprehension as being an enabler for significantly shorter proofs in HOL, compared to using FOL.

Handling *equality* is more challenging in HOL since it now applies to function-valued and Boolean-valued terms, and arriving at Henkin completeness requires handling the extensionality of functions and propositions. The respective axiom and scheme for *Boolean extensionality* (or *propositional extensionality*) and *functional extensionality*) are

$$\forall X^o Y.\ (X \longleftrightarrow Y) \longrightarrow X = Y$$
$$\forall F^{\tau\to\sigma} G.\ (\forall X^\tau.\ FX = GX) \longrightarrow F = G$$

[4]Andrews [1, p207] gives the Comprehension Axiom scheme as $\exists U^{\sigma\to\tau}\forall V^\sigma.\ UV = \mathbf{A}^\tau$ which when written in λ-notation shows up as the β-conversion rule.

As with equality-handling in FOL, better performance is achieved by extending a proof calculus with equality-related rules rather than adding the characterising axioms to the logic.

LEO-II does not yet support a calculus-level treatment of the axiom of choice (AC). (Choice is, however, supported in Satallax [16].) Choice is related to *Skolemization.* In HOL, Skolemization is not as straightforward as in FOL. If it were to be directly adapted for HOL, Skolemization is unsound wrt Henkin models that invalidate AC, and incomplete wrt Henkin models that validate AC [4]. That is, naïve Skolemization makes instances of AC derivable, but does not make all instances of AC derivable [8, §3.2].

LEO-II is a *resolution*-based prover. In *first-order* resolution-based theorem-proving, *clause normalisation* is only carried out once at the beginning of the process. In higher-order theorem-proving, clause normalisation might be carried out several times (at different points during the proof process) since variables may be instantiated with formulas, and this may turn normal clauses into non-normal ones.

In FOL, *unification* is decidable, and it is used as an eager filter during resolution. Higher-order unification is undecidable in general, and the application of unification is more delicate. LEO-II relies on a variant of Huet's *pre-unification* [17] procedure, which is semi-decidable. It checks for the existence of unifiers without actually producing them – as pointed out by Huet, this is usually sufficient to produce refutations where possible [18]. It works by accumulating flex-flex unification pairs as unification constraints. When a clause consists only of flex-flex constraints then it is considered to be *empty*, since, as Huet showed [19], such a system of equations always has solutions.

Resolution and factorisation may be applied to the unification constraints too. Despite the theoretical benefit of lazy filtering, this produces problems in practice owing to accumulation, as described by Benzmüller [5, §3.3]. Benzmüller explains that there is a tension between performance and completeness, and advocates applying eager unification (with a depth bound) despite that this makes the system incomplete. Though it was originally intended as an alternative option for LEO-II's architecture, lazy unification has not yet been implemented. Eager unification in LEO-II works as follows: pre-unification is applied to clauses with a predefined depth bound (e.g. maximally five[5] nestings of the branching FLEXRIGID rule; cf.§4.3). The solved unification constraints are exhaustively applied (with rule SUBST; cf.§4.3) in the resulting clauses, and any remaining flex-flex unification pairs are kept as unification constraints of the result clause. Pre-unification may return an empty clause – that is, a clause which is either literally empty or which consists only of flex-flex unification constraints, which always have a solution.

Unification is used to find instantiations of variables of arbitrary type. In higher-order theorem-proving, an additional form of instantiation is required for completeness. This form of instantiation only concerns predicate variables. For example, in order to prove $\exists P.P$ or $\exists P \exists X.P\,X$ we cannot use unification. This form of instantiation is called *primitive substitution.* Guessing instantiations for such variables is a comprehensive challenge since the search is infinitely-branching. Whereas in FOL one can have a complete resolution calculus using only the factorisation and resolution rules, in higher-order resolution we need an additional rule for primitive substitution.

[5]The pre-unification depth is a parameter in LEO-II that can be specified at the command line. By default LEO-II currently operates with values up to depth 8. So far there has been no exhaustive empirical investigation of the optimal setting of the pre-unification depth.

3.2 More on LEO-II

Leo-II [15] is implemented in OCaml and implements several improvements over the original Leo prover. Leo-II's calculus is based on RUE resolution: in Leo-II, unification constraints are disagreement pairs, and are amenable to resolution. Leo-II supports primitive equality handling – in contrast with Leo which expanded equality using the Leibniz definition. Use of primitive equality also facilitates the link-up with first-order theorem-provers since Leibniz equality is not a first-order definition, and the target prover usually has an optimised handling of equality. The original Leo used Leibniz equality but in order to link it up with a first-order theorem-prover an intermediate primitive-equality representation was used [13, §3.3]; the intermediate representation was made available only to the first-order theorem-prover.

First-order subproofs may appear in Leo-II proofs, indicating that those subproofs were found by a first-order theorem-prover (usually E [28]) collaborating with Leo-II. Indeed, Leo-II *must* collaborate with a first-order prover since it fails to prove most problems alone. Leo had previously been interfaced with Bliksem [13]. Periodically – by default, every 10 iterations of the main reasoning loop – Leo-II submits problems to first-order provers. These problems consist of clauses which are essentially first-order. A clause is checked for its first-order character upon the clause's formation, and tagged accordingly. Leo-II supports two translations from higher-order to first-order logic, one by Kerber [21] and the other by Hurd [20]. Either translation can be used by setting a flag. Hurd's translation is used by default.

Earlier work showed that using a single higher-order strategy, when a higher-order prover is combined with a first-order prover, is better than using multiple higher-order strategies [13, §4]. The original Leo included multiple strategies [5], but Leo-II currently only implements the SOS strategy.

In Leo-II problems are read, and proofs written, in TPTP syntax. Further improvements over the original Leo are that terms in Leo-II are indexed and shared for more efficient storage and lookups [33], and Leo-II implements goal splitting based on the description by Nonnengart and Weidenbach [25].

4 Calculus

We break down the proof calculus used by Leo-II into five sets of rules based on their function. Leo-II's proof output will report inferences which are instances of the rules whose names are shown in sans in this section. An example of such a proof is provided in Appendix A.2. It might be useful to refer to the example proof as we go through Leo-II's calculus. Not all the inferences in a Leo-II proof can be described here – this is because the Leo-II proof may contain inferences carried out by external theorem-provers it relies on. For instance, lines 221-244 in Appendix A.2 are E inferences.

Rule labels shown in typefaces other than sans will not appear in Leo-II's proof output. Such rules are drawn from other sources for comparison, or are used by Leo-II but not reported – i.e. they form part of a more powerful Leo-II rule.

4.1 Normalisation

These rules deal with the normalisation of clauses, and this section is based on [7, p49]. Each rule name has a suffix consisting of 'pos' or 'neg', referring to the polarity of the focussed literal in that rule. The rules are straightforward – for instance, extcnf_or_pos lifts object-level disjunction to meta-level (i.e. clause-level) disjunction. Similarly, the rule extcnf_not_pos removes a dominant negation from a literal and flips the literal's polarity.

$$\frac{\mathbf{C} \vee [\mathbf{A} \vee \mathbf{B}]^{\mathtt{tt}}}{\mathbf{C} \vee [\mathbf{A}]^{\mathtt{tt}} \vee [\mathbf{B}]^{\mathtt{tt}}}\ \textsf{extcnf\_or\_pos} \qquad \frac{\mathbf{C} \vee [\neg\mathbf{A}]^{\mathtt{tt}}}{\mathbf{C} \vee [\mathbf{A}]^{\mathtt{ff}}}\ \textsf{extcnf\_not\_pos} \qquad \frac{\mathbf{C} \vee [\mathbf{A} \vee \mathbf{B}]^{\mathtt{ff}}}{\mathbf{C} \vee [\mathbf{A}]^{\mathtt{ff}} \quad \mathbf{C} \vee [\mathbf{B}]^{\mathtt{ff}}}\ \textsf{extcnf\_or\_neg}$$

$$\frac{\mathbf{C} \vee [\neg\mathbf{A}]^{\mathtt{ff}}}{\mathbf{C} \vee [\mathbf{A}]^{\mathtt{tt}}}\ \textsf{extcnf\_not\_neg} \qquad \frac{\mathbf{C} \vee [\Pi^{\tau}\mathbf{A}]^{\mathtt{tt}} \quad X^{\tau} \text{ fresh variable}}{\mathbf{C} \vee [\mathbf{A}\,X]^{\mathtt{tt}}}\ \textsf{extcnf\_forall\_pos}$$

$$\frac{\mathbf{C} \vee [\Pi^{\tau}\mathbf{A}]^{\mathtt{ff}} \quad \mathtt{sk}^{\tau} \text{ Skolem term}}{\mathbf{C} \vee [\mathbf{A}\,\mathtt{sk}^{\tau}]^{\mathtt{ff}}}\ \textsf{extcnf\_forall\_neg}$$

$$\frac{\mathbf{C} \vee [\Pi^{\tau}\mathbf{A}]^{\mathtt{tt}} \quad \text{canonical } \overline{\mathbf{B}^{\tau}}^{n} \quad n = \text{Card}(\tau) \quad n \in \omega}{\mathbf{C} \vee \overline{[\mathbf{A}\,\mathbf{B}]^{\mathtt{tt}}}^{n}}\ \textsf{extcnf\_forall\_special\_pos}$$

Rule extcnf-forall-special-pos is a special instance of the rule extcnf_forall_pos; it is used for the exhaustive instantiation of some finite types τ having cardinality n. The rule instantiates n clauses, each with a different term of type τ. Currently, this only applies when τ is o, $o \to o$ or $o \to o \to o$.

Examples of these rules' instances can be seen in Appendix A.2. For example, lines 175-180 show an instance of extcnf_or_neg (projecting the right disjunct from the hypothesis into the conclusion):

$$\frac{[(\mathrm{sK}_2 \neq \lambda \mathrm{SX}_0^{\iota}.\ \mathrm{SX}_0) \vee \neg((\lambda \mathrm{SX}_0^{\iota}.\ \mathrm{sK}_3\ (\mathrm{sK}_2\ \mathrm{SX}_0)) \neq \mathrm{sK}_3)]^{\mathtt{ff}}}{[\neg((\lambda \mathrm{SX}_0^{\iota}.\ \mathrm{sK}_3\ (\mathrm{sK}_2\ \mathrm{SX}_0)) \neq \mathrm{sK}_3)]^{\mathtt{ff}}}$$

Perhaps more interesting is an instance of extcnf_forall_neg on lines 216-220, in which the bound variable SV_3^{ι} is replaced with the Skolem constant sK_4:

$$\frac{[\forall \mathrm{SV}_3^{\iota}.\ \mathrm{sK}_3\ (\mathrm{sK}_2\ \mathrm{SV}_3)\ =\ \mathrm{sK}_3\ \mathrm{SV}_3]^{\mathtt{ff}}}{[\mathrm{sK}_3\ (\mathrm{sK}_2\ \mathrm{sK}_4)\ =\ \mathrm{sK}_3\ \mathrm{sK}_4]^{\mathtt{ff}}}$$

There are three rules which appear in LEO-II's proof output but which are not explicitly formalised above. Rule sim carries out trivial simplifications, including rewriting using idempotency identities. Rule extcnf_combined is a combination of the rules described in sections 4.1 and 4.2. Lacking more specification, the inferences that are made in an extcnf_combined step need to be rediscovered by proof search when reconstructing the proof. Rule standard_cnf refers to the use of a combination of rules shown in §4.1. This rule is only applied during a preprocessing stage while splitting.

4.2 Extensionality

$$\frac{\mathbf{C} \vee [\mathbf{M}^{\sigma\tau} = \mathbf{N}^{\sigma\tau}]^{\mathtt{tt}} \quad X^{\tau} \text{ fresh variable}}{\mathbf{C} \vee [\mathbf{M}\,X = \mathbf{N}\,X]^{\mathtt{tt}}}\ \textsc{FuncPos} \qquad \frac{\mathbf{C} \vee [\mathbf{M}^{o} = \mathbf{N}^{o}]^{\mathtt{tt}}}{\mathbf{C} \vee [\mathbf{M}^{o} \longleftrightarrow \mathbf{N}^{o}]^{\mathtt{tt}}}\ \textsc{BoolPos}$$

Starting with LEO-II version 1.2.7, rules FUNCPOS and BOOLPOS have been combined into the rule extcnf_equal_pos . The rule FUNCPOS is called *Func′* in [7, p70], where BOOLPOS is called *Equiv′*.

An example of extcnf_equal_pos can be seen on line 197 in Appendix A.2. The inference being carried out there is the following instance of extcnf_equal_pos:

$$\frac{[\mathrm{sK}_2 \ = \ \lambda \mathrm{SX}_0^{\iota}.\mathrm{SX}_0]^{\mathrm{tt}}}{[\forall \mathrm{SV}_1^{\iota}.\ \mathrm{sK}_2\ \mathrm{SV}_1 \ = \ \mathrm{SV}_1]^{\mathrm{tt}}}$$

That is, a positive equality between functions is turned into an equality between individuals.

Rule extcnf_equal_neg collects the negative Boolean and function extensionality rules shown below. These rules could be classified as unification rules (described in the next section) but it might be more consistent to describe them here together with the extensionality rules.

$$\frac{\mathbf{C} \vee [\mathbf{M}^{\sigma\tau} = \mathbf{N}^{\sigma\tau}]^{\mathrm{ff}} \quad \mathsf{sk}^{\tau} \text{ a Skolem term}}{\mathbf{C} \vee [\mathbf{M}\,\mathsf{sk} = \mathbf{N}\,\mathsf{sk}]^{\mathrm{ff}}}\ \textsc{FuncNeg} \qquad \frac{\mathbf{C} \vee [\mathbf{M}^{o} = \mathbf{N}^{o}]^{\mathrm{ff}}}{\mathbf{C} \vee [\mathbf{M}^{o} \longleftrightarrow \mathbf{N}^{o}]^{\mathrm{ff}}}\ \textsc{BoolNeg}$$

4.3 Unification

This set of rules implements pre-unification and is based on [7, p50]. LEO-II does not report the specific unification steps it makes – rather, it packages unification steps (which may include applications of rule extcnf_equal_neg described above) into a rule called extuni. The set of rules shown below differs slightly from [7, p50]: rule *Leib* is not used in LEO-II, rules *Func* and *Equiv* are collected under extcnf_equal_neg (§4.2), and rule *Dec* appears differently[6].

$$\frac{\mathbf{C} \vee [\mathbf{A} = \mathbf{A}]^{\mathrm{ff}}}{\mathbf{C}}\ \textsc{Triv} \qquad \frac{\mathbf{C} \vee [h^{\sigma\tau}\overline{\mathbf{U}^{\alpha}}^{k} \ = \ h^{\sigma\tau}\overline{\mathbf{V}^{\alpha}}^{k}]^{\mathrm{ff}}}{\mathbf{C} \vee \overline{[\mathbf{U}_i \ = \ \mathbf{V}_i]^{\mathrm{ff}}}^{i\leq k}}\ \textsc{Dec}$$

$$\frac{\mathbf{C} \vee [X = \mathbf{A}]^{\mathrm{ff}} \quad X \notin \mathrm{FV}(\mathbf{A})}{\mathbf{C}[\mathbf{A}/X]}\ \textsc{Subst} \qquad \frac{\mathbf{C} \vee [F^{\tau}\overline{\mathbf{U}}^{n} \ = \ h\overline{\mathbf{V}}^{m}]^{\mathrm{ff}} \quad \mathbf{G} \in \mathcal{AB}^{(h)}_{\tau}}{\mathbf{C} \vee [F = \mathbf{G}]^{\mathrm{ff}} \ \vee \ [F\overline{\mathbf{U}}^{n} = h\overline{\mathbf{V}}^{m}]^{\mathrm{ff}}}\ \textsc{FlexRigid}$$

In rule FLEXRIGID (and again in rule prim_subst below), we use the symbol $\mathcal{AB}^{(k)}_{\tau}$ to denote the set of *approximating/partial bindings* parametric to a type τ and to a constant k. This is explained further next, based on [7, p18], who in turn cites Snyder and Gallier [30]. Let type τ be of the form $\overline{\sigma}^{l} \to \tau'$. Given a name[7] k of type $\overline{\rho}^{l} \to \tau'$, term $\mathbf{G}$ having form $\lambda\overline{X_{\sigma^{i\leq l}}}^{l}.k\overline{\mathbf{V}}^{m}$ is a *partial binding* of type τ and *head* k. Each $\mathbf{V}^{i\leq m}$ has form $H^{i}\overline{X_{\sigma^{j\leq l}}}^{l}$ where $H^{i\leq m}$ are fresh variables typed $\overline{\sigma}^{l} \to \rho^{i\leq m}$. *Projection bindings* are partial bindings whose head k is one of $X^{i\leq l}$. *Imitation bindings* are partial binding whose head k is drawn from the set of constants in the signature. $\mathcal{AB}^{(k)}_{\tau}$ is the set of all projection and imitation bindings modulo τ and k.

[6]The decomposition rule used in LEO-II is taken from [10]. The version shown in [7, p50] is:

$$\frac{\mathbf{C} \vee [\mathbf{A}^{\sigma\tau}\overline{\mathbf{U}^{\alpha}}^{k} = \mathbf{B}^{\sigma\tau}\overline{\mathbf{V}^{\alpha}}^{k}]^{\mathrm{ff}}}{\mathbf{C} \vee [\mathbf{A} = \mathbf{B}]^{\mathrm{ff}} \vee \overline{[\mathbf{U}_i = \mathbf{V}_i]^{\mathrm{ff}}}^{i\leq k}}\ \textsc{Dec}$$

[7]where a *name* is either a constant or a variable

LEO-II may nominally use the rule flexflex to indicate that a clause consists only of flex-flex unification constraints (which always have a solution). This rule is not written explicitly here since LEO-II follows Huet in regarding such clauses to be empty.

The example proof in Appendix A.2 contains an instance of an extuni inference in lines 143-145. This inference follows the inference in clause 14 (lines 135-142) and instantiates variable SV_1 in the latter with the term $\lambda \mathrm{SX}_0^\iota.\mathrm{SX}_0$ (the identity function). This instantiation then allows us to rewrite lines 136-140 to

$$(\lambda \mathrm{SY}_{27}^\iota.\mathrm{sK}_1\ \mathrm{SY}_{27}) = (\lambda \mathrm{SY}_{28}^\iota.\mathrm{sK}_1\ \mathrm{SY}_{28})$$

which is obviously true, thus leading to $\mathtt{ff} = \mathtt{tt}$ in line 144.

4.4 Resolution

$$\frac{[\mathbf{A}]^{p_1} \vee \mathbf{C} \quad [\mathbf{B}]^{p_2} \vee \mathbf{D} \quad p_1 \neq p_2}{\mathbf{C} \vee \mathbf{D} \vee [\mathbf{A} = \mathbf{B}]^{\mathtt{ff}}}\ \mathsf{res} \qquad \frac{[\mathbf{A}]^p \vee [\mathbf{B}]^p}{[\mathbf{A}]^p \vee [\mathbf{A} = \mathbf{B}]^{\mathtt{ff}}}\ \mathsf{fac\_restr}$$

This section is based on [7, p50]. Rule res is the resolution rule and fac_restr is the restricted factorisation rule. The *full* factorisation rule is described in [7, p50], but LEO-II uses a restricted form of factorisation which targets binary clauses, as formalised above. Resolution and restricted factorisation are applied only to proper clauses in LEO-II.

We now turn to the rule prim_subst, which is called *Prim* in [7, p50]. This is the *primitive substitution* rule used to instantiate predicate variables, as mentioned in §3.1. Primitive substitution is applied only to proper clauses in LEO-II. This rule requires some context beforehand: Let Σ be the logic's signature, and $\kappa \subseteq \Sigma$ range over logical constants, i.e. $\kappa = \{\neg, \vee\} \cup \{\Pi^{(\sigma\to o)\to o}, =^{\sigma\to\sigma\to o}\colon \sigma \in \mathcal{T}\}$, where $\mathcal{T}$ is the set of types. Let $\tau, \sigma \in \mathcal{T}$ range over types, and p range over truth values $\{\mathtt{tt}, \mathtt{ff}\}$. Let Q_τ be a flexible literal head – i.e. τ must have the form $\sigma \to o$. Finally, the prim_subst rule, which simulates Huet's *splitting rule* in his resolution calculus, is formalised as follows:

$$\frac{[Q_\tau \overline{\mathbf{U}}^n]^p \vee \mathbf{C} \quad \mathbf{P} \in \mathcal{AB}_\tau^{(k)}}{([Q_\tau \overline{\mathbf{U}}^n]^p \vee \mathbf{C})[\mathbf{P}/Q]}\ \mathsf{prim\_subst}$$

As an example consider the formula $\exists P \exists X. P\, X$. Negating and normalising the formula gives the clause $[P\, X]^{\mathtt{ff}}$. Rule prim_subst offers the clause $[\neg H\, X]^{\mathtt{ff}}$ by using the instantiation $[(\lambda X. \neg H\, X)/P]$. Further normalisation and resolution will yield a singleton clause consisting of a flex-flex constraint – that is, an effectively empty clause.

4.5 Logistic

Logistic rules include rules which handle book-keeping and other, perhaps extralogical, operations. We start with splitting. Rules split_conjecture and solved_all_splits form a proof scheme; they can be described together as shown below. In this scheme, note that $(\circ, p) \in \{(\wedge, \mathtt{ff}), (\vee, \mathtt{tt})\}$.

$$\dfrac{\dfrac{\mathbf{C} \vee [\mathbf{A}_1 \circ \ldots \circ \mathbf{A}_n]^p}{\dfrac{\mathbf{C} \vee [\mathbf{A}_1]^p}{\dfrac{\vdots}{[\mathtt{ff}]^{\mathtt{tt}}}} \qquad \dfrac{\mathbf{C} \vee [\mathbf{A}_n]^p}{\dfrac{\vdots}{[\mathtt{ff}]^{\mathtt{tt}}}}}\ \mathsf{split\_conjecture} \quad \ldots}{[\mathtt{ff}]^{\mathtt{tt}}}\ \mathsf{solved\_all\_splits}$$

The rule scheme could also be used to split on $(\Leftrightarrow, \mathtt{ff})$ and $(=, \mathtt{ff})$, both of which are used for equality over propositions. For an example of this rule's use, refer to clause 3 (lines 44-58) in Appendix A.2, and how it splits into clauses 4 and 5.

Rule negate_conjecture is another example of a proof scheme. It is used to build refutation proofs. It formalises the contradiction proof method: the conjecture is negated at the start of the proof, then the proof machinery attempts to refute the negated conjecture to prove the original conjecture valid. An instance of negate_conjecture can be seen in Appendix A.2, in lines 33-43.

$$\dfrac{\dfrac{\dfrac{\neg\mathbf{A}}{\vdots}\text{negate\_conjecture}}{[\mathtt{ff}]^{\mathtt{tt}}}}{\mathbf{A}}$$

Rule polarity_switch flips the polarity of literals and negates their formulas. (Note that extcnf_not_pos and extcnf_not_neg, seen in §4.1, only remove negations.) As one would expect, this rule is only used in controlled settings to avoid nontermination. Let $\bar{\mathtt{tt}} = \mathtt{ff}$ and $\bar{\mathtt{ff}} = \mathtt{tt}$, then:

$$\frac{\mathbf{C} \vee [\mathbf{A}]^p}{\mathbf{C} \vee [\neg\mathbf{A}]^{\bar{p}}}\ \text{polarity\_switch}$$

The inference fo_atp_e indicates that a subproof, relating to a particular formula in the proof, was found using the E theorem prover. The formula in question appears as the conclusion of the fo_atp_e inference; the inference also specifies the hypotheses from which the conclusion was deduced. E's proof is spliced back into LEO-II's proof graph, as shown in the example in the appendix. This splicing requires LEO-II to carry out several manipulations on the proof delivered by E. For example, a renaming of clauses is required to avoid clashes (clauses, representing proof steps in proofs, are usually given numbers as names – and thus LEO-II must ensure that all steps in the combined LEO-II+E proof are unique). Moreover, the correspondence between the input clauses given to E and the originating clauses in LEO-II has to be established. This is illustrated in Appendix A.2 by clauses 32 (lines 221-223) and 33: their annotation has been modified by LEO-II such that a reference to the corresponding LEO-II clauses 27 an 30 is made, together with the justification FULLY-TYPED-TRANSLATION.

Rules unfold_def, rename and copy carry out straightforward administrative tasks. Rule unfold_def expands *definitions* – these are specially-marked equational axioms, of the form $h^\tau = \mathbf{A}^\tau$, where the LHS is a constant which does not appear on the RHS, and the constant does not appear as the sole LHS in another definition. For an example of this rule in action, compare lines 34-42 with lines 45-57 in Appendix A.2.

Rule rename simply renames all free variables in a clause, and rule copy merely copies a formula from a prior inference. This rule allows LEO-II to 'forget' annotation information associated with the original formula – this annotation information is used as heuristic information to help guide proof-search.

5 Proof procedure

LEO-II uses a DISCOUNT-based [3] given-clause reasoning loop and terminates upon generating an empty clause, or after exceeding a timeout or maximum number of iterations. During processing, clauses are checked for being first-order, in which case they are labelled with a flag indicating such.

5.1 Preprocessing

Before starting the main reasoning loop the following preprocessing steps are carried out in the order shown below.

Abbreviation expansion involves expanding non-logical definitions (i.e. Boolean-valued definitions, such as $\wedge$ or $\longrightarrow$, are not touched). Expansion of definitions is exhaustively applied – see explanation of rule unfold_def in §4.5. Recursive definitions cause LEO-II to diverge.

Standard normalisation involves the application of clause normalisation rules but does not use extensionality rules.

Splitting involves breaking up formulas dominated by connectives such as $\wedge$ or $\longleftrightarrow$ into separate problems – see the explanation of rule split_conjecture in §4.5. Each subproblem is then tackled as follows:

1. **Abbreviation expansion** is applied, but this time logical definitions may be expanded (e.g. occurrences of $\wedge$ or $\longrightarrow$);
2. Each clause is **extensionally normalised** (this may include Skolemization) using the extcnf-combined family of rules (§4.1);
3. **Primitive substitution** (rule prim_subst described in §4.4) is used to instantiate predicate variables;
4. **Restricted factorization** (rule fac_restr described in §4.4);
5. **Positive functional extensionality and positive Boolean extensionality** (rule extcnf_equal_pos described in §4.2);
6. **Depth-bounded extensional pre-unification** (which may include Skolemization) appears as rule extuni and is described in §4.3;
7. **Extensional normalisation** (which may include Skolemization too) appears as rule extcnf_combined and is described in §4.1;
8. Finally clauses are **simplified** (rule sim described in §4.1).

Initialisation The *active* clauseset is initialised with the output from the above step, and *passive* clauseset is initialised to the empty set.

5.2 Reasoning loop

1. At regular intervals, dispatch first-order clauses to the external prover. The first-order prover is called on essentially first-order clauses which have been recognised up to this

part of the process.[8] If LEO-II is given a first-order problem then it is passed on to E to solve at this point. Conclusions of refutations found by E are labelled by an fo-atp-e inference, as described in §4.5.

2. Select a clause from the active set and rename its free variables to ensure freshness. The selection function implements the *ratio strategy* [22]: the clause's weight is the dominant criterion, and the clause's age (i.e. its identifier number) is a secondary criterion. Lighter clauses are preferred over heavy ones, and older clauses over younger ones. The weight of a clause is (currently) simply taken as the number of its literals (for future versions of LEO-II it is planned to employ proper term weightings to obtain a more meaningful weighting criterion).
3. If the selected clause is subsumed by a clause within the passive set then ignore the selected clause. Otherwise:
   (a) Add the selected clause to the passive set, and remove subsumed clauses from the active set.
   (b) Apply the following rules to the selected clause:
      - resolution (rule res, §4.4) with all members of the active set;
      - restricted factorisation (rule fac_restr, §4.4);
      - primitive substitution (rule prim_subst, §4.4);
      - positive Boolean extensionality (rule BOOLPOS, §4.2).

   (c) Apply the following rules to the resulting clauses:
      - extensional normalisation (rule extcnf_combined, §4.1);
      - depth-bound extensional pre-unification (rule extuni, §4.3);
      - simplification (rule sim, §4.1).

   (d) Add the resulting clauses to the active set.

6 Conclusion

In this paper we sought to give more logical details of LEO-II than had previously been published [15; 14]. It is hoped that this can be useful for interpreting LEO-II proofs, perhaps for subsequent use or verification by other systems.

Acknowledgements

Larry Paulson hosted the second author during a stay at Cambridge, during which development of LEO-II began. Paulson is also the research supervisor of the first author. The initial work on LEO-II was supported by the UK Engineering and Physical Sciences Research Council (grant number EP/D070511/1). Subsequent development of the system was supported by the German Research Foundation under grants BE 2501/6-1 and BE 2501/8-1.

Geoff Sutcliffe, together with the second author, has initiated the development of the TPTP THF infrastructure. This work was funded by the European Community's Seventh Framework Programme FP7/2007-2013, under grant agreement PIIF-GA-2008-219982. Sutcliffe has also been a close collaborator and advisor wrt LEO-II's TSTP proof output.

We also thank the anonymous reviewers for their feedback.

[8]Whenever a new clause is formed in LEO-II, then LEO-II's criterion for being 'essentially first-order' is checked and the new clause is annotated accordingly.

References

[1] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Kluwer Academic Publishers, 2002. ISBN 1402007639.

[2] Peter B. Andrews, Matt Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A Theorem-Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16(3): 321–353, 1996.

[3] Jürgen Avenhaus, Jörg Denzinger, and Matthias Fuchs. DISCOUNT: A system for distributed equational deduction. In Jieh Hsiang, editor, *Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 397–402. Springer Berlin / Heidelberg, 1995.

[4] Julian Backes and Chad E. Brown. Analytic tableaux for higher-order logic with choice. *Journal of Automated Reasoning*, 47(4):451–479, 2011.

[5] Christoph Benzmüller. A calculus and a system architecture for extensional higher-order resolution. Research Report 97-198, Department of Mathematical Sciences, Carnegie Mellon University, Pittsburgh, USA, 1997.

[6] Christoph Benzmüller. Extensional higher-order paramodulation and RUE-resolution. In Harald Ganzinger, editor, *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, number 1632 in LNCS, pages 399–413. Springer, 1999.

[7] Christoph Benzmüller. *Equality and Extensionality in Higher-Order Theorem Proving.* PhD thesis, Naturwissenschaftlich-Technische Fakultät I, Saarland University, Saarbrücken, Germany, 1999.

[8] Christoph Benzmüller and Chad E. Brown. A Structured Set of Higher-Order Problems. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, number 3603 in LNCS, pages 66–81. Springer, 2005.

[9] Christoph Benzmüller and Manfred Kerber. A lost proof. In *Proceedings of the IJCAR 2001 Workshop: Future Directions in Automated Reasoning*, pages 13–24, Siena, Italy, 2001.

[10] Christoph Benzmüller and Michael Kohlhase. Extensional higher-order resolution. In Claude Kirchner and Hélène Kirchner, editors, *Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings*, number 1421 in LNAI, pages 56–71. Springer, 1998.

[11] Christoph Benzmüller and Michael Kohlhase. LEO – a higher-order theorem prover. In Claude Kirchner and Hélène Kirchner, editors, *Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings*, number 1421 in LNCS, pages 139–143. Springer, 1998.

[12] Christoph Benzmüller, Chad Brown, and Michael Kohlhase. Higher-order semantics and extensionality. *Journal of Symbolic Logic*, 69(4):1027–1088, 2004.

[13] Christoph Benzmüller, Volker Sorge, Mateja Jamnik, and Manfred Kerber. Can a higher-order and a first-order theorem prover cooperate? In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005, Proceedings*, number 3452 in LNCS, pages 415–431. Springer, 2005.

[14] Christoph Benzmüller, Lawrence C. Paulson, Frank Theiss, and Arnaud Fietzke. Progress report on LEO-II – an automatic theorem prover for higher-order logic. In *TPHOLs 2007 Emerging Trends Proceedings*, pages 33–48. Internal Report 364/07, Department of Computer Science, University Kaiserslautern, Germany, 2007.

[15] Christoph Benzmüller, Frank Theiss, Larry Paulson, and Arnaud Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *LNCS*, pages 162–170. Springer, 2008.

[16] Chad E. Brown. Reducing Higher-Order Theorem Proving to a Sequence of SAT Problems. In Nikolaj Bjørner and Viorica Sofronie-Stockkermans, editors, *CADE – the 23rd International Conference on Automated Deduction*, LNCS/LNAI 6803, pages 147 – 161. Springer, July 2011.

[17] Gilles Dowek. Higher-order Unification and Matching. In *Handbook of Automated Reasoning*, pages 1009–1062. Elsevier Science Publishers BV, 2001. ISBN 0444508120.

[18] Gerard Huet. A Complete Mechanization of Type Theory. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* , pages 139–146, 1973.

[19] Gerard Huet. A Unification Algorithm for Typed Lambda-Calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[20] Joe Hurd. An LCF-style interface between HOL and first-order logic. *Automated Deduction—CADE-18*, pages 1–41, 2002.

[21] Manfred Kerber. On the translation of higher-order problems into first-order logic. In Tony Cohn, editor, *Proceedings of the 11th ECAI*, pages 145–149, Amsterdam, The Netherlands, August 1994. John Wiley & Sons, Chichester, England.

[22] W. McCune and L. Wos. Experiments in Automated Deduction with Condensed Detachment. *Automated Deduction—CADE-11*, pages 209–223, 1992.

[23] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set, HETS. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 519–522, 2007.

[24] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Verlag, 2002.

[25] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. *Handbook of Automated Reasoning*, 1:335–367, 2001.

[26] Adam Pease. *Ontology: A Practical Guide.* Articulate Software Press, 2011. ISBN: 1-889455-105.

[27] Adam Pease and Christoph Benzmüller. Sigma: An integrated development environment for formal ontology. *AI Communications (Special Issue on Intelligent Engineering Techniques for Knowledge Bases)*, 2011. To appear.

[28] Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[29] Jörg Siekmann, Christoph Benzmüller, and Serge Autexier. Computer supported mathematics with OMEGA. *Journal of Applied Logic*, 4(4):533–559, 2006.

[30] Wayne Snyder and Jean Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(1-2):101–140, 1989.

[31] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[32] Geoff Sutcliffe and Christoph Benzmüller. Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure. *Journal of Formalized Reasoning*, 3(1):1, 2010. ISSN 1972-5787.

[33] Frank Theiss and Christoph Benzmüller. Term indexing for the LEO-II prover. In *IWIL-6 workshop at LPAR 2006: The 6th International Workshop on the Implementation of Logics*, Pnom Penh, Cambodia, 2006.

[34] Steven Trac, Yury Puzis, and Geoff Sutcliffe. An Interactive Derivation Viewer. *Electronic Notes on Theoretical Computer Science*, 174:109–123, May 2007. ISSN 1571-0661.

A An example LEO-II proof

A.1 Simple example problem

The THF (see §2.2 for a brief outline of the syntax) script below starts by giving the type and definition of the composition functional over elements of type `$i` – this denotes the type of individuals in THF. The script then gives the type and definition of the identity function over individuals. It then uses these definitions to conjecture a formula labelled `thm`. This formula conjectures two properties: there exists a function F such that for any function G, $F \circ G = G \circ F$; and for all functions F and G, if F is the identity function then $F \circ G = G$. The proof[9] found by LEO-II and E for this formula is shown in the next section.

```
thf(compose_type,type,(
    compose: ( $i > $i ) > ( $i > $i ) > $i > $i )).

thf(compose_def,definition,
    ( compose
    = ( ^ [F: $i > $i,G: $i > $i,X: $i] :
          ( G @ ( F @ X ) ) ) )).

thf(id_type,type,(
    id: $i > $i )).

thf(id_def,definition,
    ( id
    = ( ^ [X: $i] : X ) )).

thf(thm,theorem,
    ( ? [F: $i > $i] :
      ! [G: $i > $i] :
        ( ( compose @ F @ G )
        = ( compose @ G @ F ) )
    & ! [F: $i > $i,G: $i > $i] :
        ( ( F = id )
       => ( ( compose @ F @ G )
          = G ) ) )).
```

A.2 A joint LEO-II+E proof for the example problem

This TPTP script is a proof in higher-order logic but contains a first-order proof segment which was found by E. The parts contributed by E can easily be identified by the TPTP *language* (see §2.2) of the proof's clauses: those in FOF and CNF are contributed by E. The contradiction found using FOL is used to finish the containing HOL proof.

Let us quickly outline the structure of lines in the proof, taking the following as an example:

[9]To obtain detailed proof information, as illustrated here, from LEO-II you must give LEO-II (versions 1.3 and higher) the argument '-po 2' ('-po' stands for *proof output*). The E subproof is not shown when LEO-II is called with the '-po 1' or '-po 0' options.

```
1  thf(19,plain,
2      ( ( ( sK2
3         != ( ^ [SX0: $i] : SX0 ) )
4        | ~ ( ( ^ [SX0: $i] :
5                   ( sK3 @ ( sK2 @ SX0 ) ) )
6           != sK3 ) )
7      = $false ),
8      inference(extcnf_not_pos,[status(thm)],[18])).
```

Line 1 tells us that the formula carried by the line is written in THF, that the line's label (think of this as a unique name) is "19", and that its role is plain (i.e. it's not an axiom, lemma, definition, etc – see the TPTP Technical Report[10] for a full list of roles). Lines 2-7 state the THF formula [32] being concluded by this inference. Line 8 consists of an annotation providing valuable proof information: it tells us that the inference is an instance of extcnf_not_pos, that the inference is validity-preserving, and that the rule's hypothesis is identified by the label "18". Rule extcnf_not_pos is described in §4.1. Clause 18, on which the inference described above relies, is shown below – the boxes contain the parts of its formula affected by extcnf_not_pos.

```
thf(18,plain,
    ( ( ~ ( ( sK2
          != ( ^ [SX0: $i] : SX0 ) )
         | ~ ( ( ^ [SX0: $i] :
                   ( sK3 @ ( sK2 @ SX0 ) ) )
            != sK3 ) ) )
    = $true ),
    inference(unfold_def,[status(thm)],[17,compose,id])).
```

The full LEO-II+E proof for the problem described in Appendix A.1 is given next.

```
% SZS status Theorem for /tmp/SystemOnTPTPFormReply928/SOT_B5Kj_i (rf:0,ps:3,sos:true,u:8,ude:true,foatp:e)
%**** Beginning of derivation protocol ****
% SZS output start CNFRefutation
thf(tp_compose,type,(
    compose: ( $i > $i ) > ( $i > $i ) > $i > $i )).
thf(tp_id,type,(
    id: $i > $i )).
thf(tp_sK1,type,(
    sK1: ( $i > $i ) > $i > $i )).
thf(tp_sK2,type,(
    sK2: $i > $i )).
thf(tp_sK3,type,(
    sK3: $i > $i )).
thf(tp_sK4,type,(
    sK4: $i )).
thf(compose_def,definition,
    ( compose
    = ( ^ [F: $i > $i,G: $i > $i,X: $i] :
          ( G @ ( F @ X ) ) ) )).
thf(id_def,definition,
    ( id
    = ( ^ [X: $i] : X ) )).
thf(1,conjecture,
    ( ? [F: $i > $i] :
      ! [G: $i > $i] :
        ( ( compose @ F @ G )
        = ( compose @ G @ F ) )
    & ! [F: $i > $i,G: $i > $i] :
        ( ( F = id )
```

[10]http://www.cs.miami.edu/~tptp/TPTP/TR/TPTPTR.shtml

```
       => ( ( compose @ F @ G )
          = G ) ) ),
    file('/tmp/SystemOnTPTPFormReply928/SOT_B5Kj_i',thm)).
thf(2,negated_conjecture,
    ( ( ? [F: $i > $i] :
        ! [G: $i > $i] :
          ( ( compose @ F @ G )
          = ( compose @ G @ F ) )
      & ! [F: $i > $i,G: $i > $i] :
          ( ( F = id )
         => ( ( compose @ F @ G )
            = G ) ) )
    = $false ),
    inference(negate_conjecture,[status(cth)],[1])).
thf(3,plain,
    ( ( ? [SY0: $i > $i] :
        ! [SY1: $i > $i] :
          ( ( ^ [SY4: $i] :
                ( SY1 @ ( SY0 @ SY4 ) ) )
          = ( ^ [SY7: $i] :
                ( SY0 @ ( SY1 @ SY7 ) ) ) )
      & ! [SY8: $i > $i,SY9: $i > $i] :
          ( ( SY8
            = ( ^ [X: $i] : X ) )
         => ( ( ^ [SY12: $i] :
                  ( SY9 @ ( SY8 @ SY12 ) ) )
            = SY9 ) ) )
    = $false ),
    inference(unfold_def,[status(thm)],[2,compose,id])).
thf(4,plain,
    ( ( ? [SY0: $i > $i] :
        ! [SY1: $i > $i] :
          ( ( ^ [SY4: $i] :
                ( SY1 @ ( SY0 @ SY4 ) ) )
          = ( ^ [SY7: $i] :
                ( SY0 @ ( SY1 @ SY7 ) ) ) ) )
    = $false ),
    inference(split_conjecture,[split_conjecture(split,[])],[3])).
thf(5,plain,
    ( ( ! [SY8: $i > $i,SY9: $i > $i] :
          ( ( SY8
            = ( ^ [X: $i] : X ) )
         => ( ( ^ [SY12: $i] :
                  ( SY9 @ ( SY8 @ SY12 ) ) )
            = SY9 ) ) )
    = $false ),
    inference(split_conjecture,[split_conjecture(split,[])],[3])).
thf(6,plain,
    ( ( ~ ( ? [SY0: $i > $i] :
            ! [SY1: $i > $i] :
              ( ( ^ [SY4: $i] :
                    ( SY1 @ ( SY0 @ SY4 ) ) )
              = ( ^ [SY7: $i] :
                    ( SY0 @ ( SY1 @ SY7 ) ) ) ) ) )
    = $true ),
    inference(polarity_switch,[status(thm)],[4])).
thf(7,plain,
    ( ( ~ ( ! [SY8: $i > $i,SY9: $i > $i] :
              ( ( SY8
                = ( ^ [X: $i] : X ) )
             => ( ( ^ [SY12: $i] :
                      ( SY9 @ ( SY8 @ SY12 ) ) )
                = SY9 ) ) ) )
    = $true ),
    inference(polarity_switch,[status(thm)],[5])).
thf(8,plain,
    ( ( ! [SY0: $i > $i] :
          ( ( ^ [SY21: $i] :
                ( sK1 @ SY0 @ ( SY0 @ SY21 ) ) )
         != ( ^ [SY22: $i] :
                ( SY0 @ ( sK1 @ SY0 @ SY22 ) ) ) ) )
    = $true ),
    inference(extcnf_combined,[status(esa)],[6])).
thf(9,plain,
    ( ( ( sK2
        = ( ^ [X: $i] : X ) )
      & ( ( ^ [SY26: $i] :
              ( sK3 @ ( sK2 @ SY26 ) ) )
       != sK3 ) )
    = $true ),
    inference(extcnf_combined,[status(esa)],[7])).
thf(10,plain,
    ( ( ! [SY0: $i > $i] :
          ( ( ^ [SY21: $i] :
                ( sK1 @ SY0 @ ( SY0 @ SY21 ) ) )
         != ( ^ [SY22: $i] :
                ( SY0 @ ( sK1 @ SY0 @ SY22 ) ) ) ) )
    = $true ),
    inference(copy,[status(thm)],[8])).
thf(11,plain,(
    ! [SV1: $i > $i] :
      ( ( ( ^ [SY27: $i] :
              ( sK1 @ SV1 @ ( SV1 @ SY27 ) ) )
       != ( ^ [SY28: $i] :
              ( SV1 @ ( sK1 @ SV1 @ SY28 ) ) ) )
      = $true ) ),
    inference(extcnf_forall_pos,[status(thm)],[10])).
thf(12,plain,(
    ! [SV1: $i > $i] :
      ( ( ( ^ [SY27: $i] :
              ( sK1 @ SV1 @ ( SV1 @ SY27 ) ) )
        = ( ^ [SY28: $i] :
              ( SV1 @ ( sK1 @ SV1 @ SY28 ) ) ) )
      = $false ) ),
    inference(extcnf_not_pos,[status(thm)],[11])).
thf(14,plain,(
    ! [SV1: $i > $i] :
      ( ( ( ^ [SY27: $i] :
              ( sK1 @ SV1 @ ( SV1 @ SY27 ) ) )
        = ( ^ [SY28: $i] :
              ( SV1 @ ( sK1 @ SV1 @ SY28 ) ) ) )
      = $false ) ),
    inference(extcnf_equal_neg,[status(thm)],[12])).
thf(16,plain,(
    $false = $true ),
    inference(extuni,[status(esa)],[14:[bind(SV1,$thf(^ [SX0: $i] : SX0))]])).
thf(17,plain,
    ( ( ( sK2
        = ( ^ [X: $i] : X ) )
      & ( ( ^ [SY26: $i] :
              ( sK3 @ ( sK2 @ SY26 ) ) )
       != sK3 ) )
    = $true ),
    inference(copy,[status(thm)],[9])).
thf(18,plain,
    ( ( ~ ( ( sK2
           != ( ^ [SX0: $i] : SX0 ) )
          | ~ ( ( ^ [SX0: $i] :
                    ( sK3 @ ( sK2 @ SX0 ) ) )
             != sK3 ) ) )
    = $true ),
    inference(unfold_def,[status(thm)],[17,compose,id])).
thf(19,plain,
    ( ( ( sK2
       != ( ^ [SX0: $i] : SX0 ) )
      | ~ ( ( ^ [SX0: $i] :
                ( sK3 @ ( sK2 @ SX0 ) ) )
         != sK3 ) )
    = $false ),
    inference(extcnf_not_pos,[status(thm)],[18])).
thf(20,plain,
    ( ( sK2
     != ( ^ [SX0: $i] : SX0 ) )
    = $false ),
    inference(extcnf_or_neg,[status(thm)],[19])).
thf(21,plain,
    ( ( ~ ( ( ^ [SX0: $i] :
              ( sK3 @ ( sK2 @ SX0 ) ) )
         != sK3 ) )
    = $false ),
    inference(extcnf_or_neg,[status(thm)],[19])).
thf(22,plain,
    ( ( sK2
      = ( ^ [SX0: $i] : SX0 ) )
    = $true ),
    inference(extcnf_not_neg,[status(thm)],[20])).
thf(23,plain,
    ( ( ( ^ [SX0: $i] :
          ( sK3 @ ( sK2 @ SX0 ) ) )
     != sK3 )
    = $true ),
    inference(extcnf_not_neg,[status(thm)],[21])).
thf(24,plain,
    ( ( ! [SV1: $i] :
         ( ( sK2 @ SV1 )
         = SV1 ) )
    = $true ),
    inference(extcnf_equal_pos,[status(thm)],[22])).
thf(26,plain,
    ( ( ( ^ [SX0: $i] :
          ( sK3 @ ( sK2 @ SX0 ) ) )
      = sK3 )
    = $false ),
    inference(extcnf_not_pos,[status(thm)],[23])).
thf(27,plain,(
    ! [SV2: $i] :
      ( ( ( sK2 @ SV2 )
        = SV2 )
      = $true ) ),
    inference(extcnf_forall_pos,[status(thm)],[24])).
thf(28,plain,
    ( ( ! [SV3: $i] :
         ( ( sK3 @ ( sK2 @ SV3 ) )
         = ( sK3 @ SV3 ) ) )
    = $false ),
    inference(extcnf_equal_neg,[status(thm)],[26])).
thf(30,plain,
    ( ( ( sK3 @ ( sK2 @ sK4 ) )
      = ( sK3 @ sK4 ) )
    = $false ),
    inference(extcnf_forall_neg,[status(esa)],[28])).
fof(32,axiom,(
    ! [X1] : ti(at(ti(sK2,ft(i,i)),ti(X1,i)),i) = ti(X1,i) ),
    inference(fully_typed_translation,[status(thm)],[27])).
fof(33,axiom,(
    ti(at(ti(sK3,ft(i,i)),ti(at(ti(sK2,ft(i,i)),ti(sK4,i)),i)),i) != ti(at(ti(sK3,ft(i,i)),ti(sK4,i)),i) ),
    inference(fully_typed_translation,[status(thm)],[30])).
fof(35,plain,(
    ti(at(ti(sK3,ft(i,i)),ti(at(ti(sK2,ft(i,i)),ti(sK4,i)),i)),i) != ti(at(ti(sK3,ft(i,i)),ti(sK4,i)),i) ),
    inference(fof_simplification,[status(thm)],[33])).
fof(38,plain,(
    ! [X2] : ti(at(ti(sK2,ft(i,i)),ti(X2,i)),i) = ti(X2,i) ),
    inference(variable_rename,[status(thm)],[32])).
cnf(39,plain,
    ( ti(at(ti(sK2,ft(i,i)),ti(X1,i)),i) = ti(X1,i) ),
    inference(split_conjunct,[status(thm)],[38])).
cnf(40,plain,
    ( ti(at(ti(sK3,ft(i,i)),ti(at(ti(sK2,ft(i,i)),ti(sK4,i)),i)),i) != ti(at(ti(sK3,ft(i,i)),ti(sK4,i)),i) ),
    inference(split_conjunct,[status(thm)],[35])).
cnf(42,plain,
    ( $false ),
    inference(rw,[status(thm)],[40,39,theory(equality)])).
cnf(43,plain,
    ( $false ),
    inference(cn,[status(thm)],[42,theory(equality)])).
thf(44,plain,(
    $false = $true ),
    inference(fo_atp_e,[status(thm)],[43])).
thf(45,plain,(
    $false ),
    inference(solved_all_splits,[solved_all_splits(join,[])],[44,16])).
% SZS output end CNFRefutation
%**** End of derivation protocol ****
%**** no. of clauses in derivation: 36 ****

% END OF SYSTEM OUTPUT
% RESULT: SOT_B5Kj_i - LEO-II---1.3.0 says Unsatisfiable - CPU = 0.03 WC = 0.06
% OUTPUT: SOT_B5Kj_i - LEO-II---1.3.0 says CNFRefutation - CPU = 0.03 WC = 0.06
```

A.3 Visualising a LEO-II+E proof

The figure on the next page is an ad-hoc visualisation based on the output from Sutcliffe's Interactive Derivation Viewer (IDV) [34] – a tool for visualising TPTP proofs.

This visualisation illustrates the mixed character of the proof: the pentagonical node at the top is the original conjecture (clause 1) which is subsequently negated (clause 2, appearing as the downward-pointing pentagon). The non-logical definitions 'compose' and 'id' are then unfolded (clause 3). Clause 3 is finally refuted at the bottom-right of the graph, by the derivation of the empty clause (clause 45, which appears as the double-circular node at the bottom of the graph).

The refutation of clause 3 is split in our example into two branches, both ending with contradictions (clauses 16 and 44). The right branch, which consists of the clauses 4, 6, 8, 10-12, 14 and 16, is a pure THF branch with a pre-unification step at the end. The grey circular nodes indicate that the clauses they represent are written in THF.

The left branch of the split is a mixed higher-order–first-order proof as indicated by the mixture of node borders and fills: grey circular nodes indicate THF clauses (clauses 5, 7, 9, 17-24, 26-28, 30); circular nodes having dashed borders stand for FOF clauses (clauses 35 and 38); circular nodes having dotted borders indicate CNF clauses (clauses 39 and 40). The triangles indicate the FOF 'axiom' (or better, 'input') clauses used by E (clauses 32 and 33); they are linked to corresponding THF clauses generated by Leo-II (clauses 27 and 30). The double-circle nodes are empty clauses. Clauses 42 and 43 are such clauses generated by E, the last of which is referenced again by a contradiction in Leo-II (clause 44). This closes the proof branch. The two branches are then combined to close the proof in node 45.

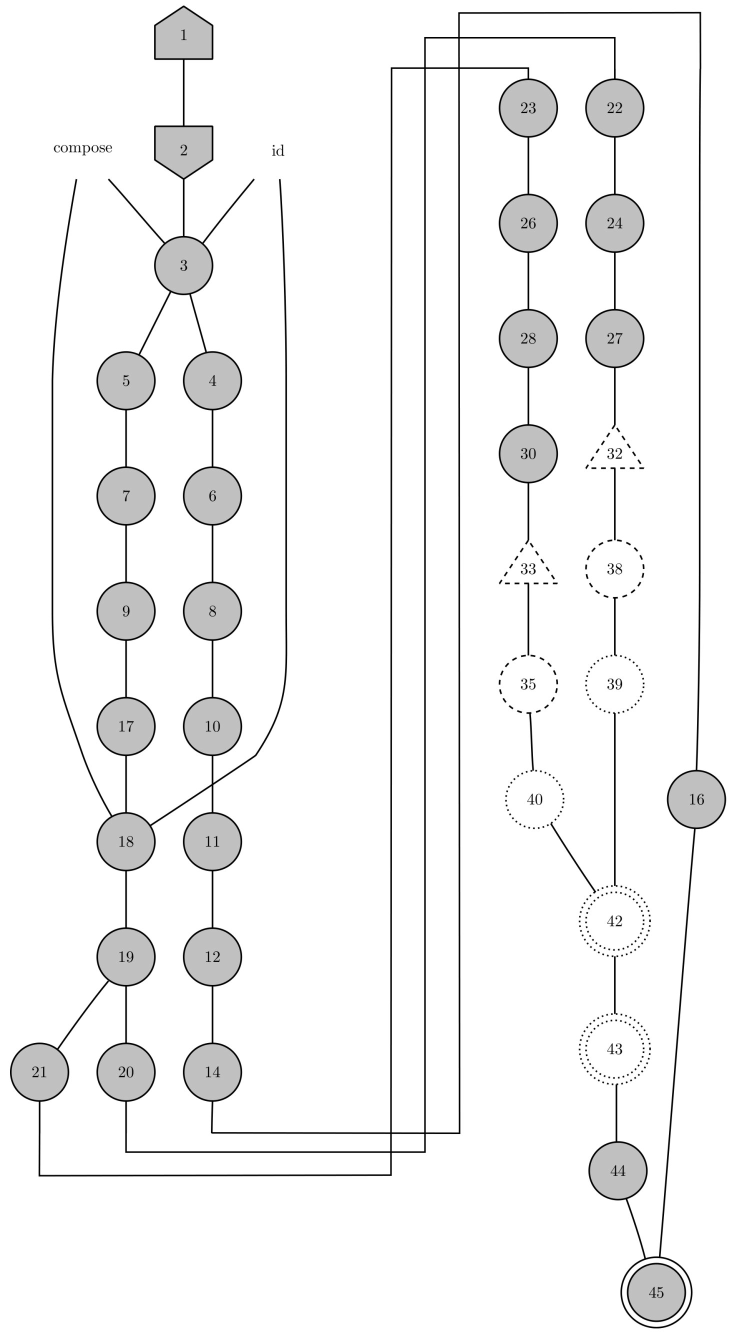

Figure 1: The proof from Appendix A.2