Kalpa Publications in Computing

Volume 1, 2017, Pages 11–26

LPAR-21S: IWIL Workshop
and LPAR Short Presentations

Leo-III Version 1.1 (System description)*

Christoph Benzmüller[21], Alexander Steen[1], and Max Wisniewski[1]

[1] Freie Universität Berlin, Berlin, Germany
c.benzmueller|a.steen|m.wisniewski@fu-berlin.de
[2] University of Luxembourg, Luxembourg

Abstract

Leo-III is an automated theorem prover for (polymorphic) higher-order logic which supports all common TPTP dialects, including THF, TFF and FOF as well as their rank-1 polymorphic derivatives. It is based on a paramodulation calculus with ordering constraints and, in tradition of its predecessor LEO-II, heavily relies on cooperation with external first-order theorem provers. Unlike LEO-II, asynchronous cooperation with typed first-order provers and an agent-based internal cooperation scheme is supported. In this paper, we sketch Leo-III's underlying calculus, survey implementation details and give examples of use.

1 Introduction

Leo-III is an automated theorem prover (ATP) for classical higher-order logic (HOL) with Henkin semantics and choice. It is the successor of the well-known LEO-II prover [8], whose development significantly influenced the build-up of the TPTP THF infrastructure [32]. Leo-III exemplarily utilizes and instantiates the associated LEOPARD system platform [36] for higher-order (HO) deduction systems implemented in Scala. The prover makes use of LEOPARD's sophisticated data structures and implements its own reasoning logic on top, inter alia as agents in LEOPARD's provided blackboard architecture. A dedicated internal reasoning agent realizes a stand-alone sequential proof procedure similar to the *given clause algorithm* of E [27]. Several other agents serve as specialists for certain tasks and can, for example, bundle multiple calculus rules that are often used in combination (cf. §5.3). Within Leo-III, agents are run in parallel controlled by a scheduler based on an optimization algorithm for combinatorical auctions.

In the spirit of the cooperative nature of LEO-II, Leo-III version 1.1 supports collaboration during proof search with external theorem provers, in particular, with first-order (FO) ATPs. Unlike LEO-II, which translated proof obligations into untyped first-order formulae, Leo-III, by default, translates its HO clauses to (polymorphically and monomorphically) typed first-order clauses. The prover thus exploits the comparably novel support for types in first-order theorem provers and, simultaneously, reduces clutter during FO encoding.

*This work has been supported by the DFG under grant BE 2501/11-1 (Leo-III).

T.Eiter, D.Sands, G.Sutcliffe and A.Voronkov (eds.), LPAR-21S (Kalpa Publications in Computing, vol. 1), pp. 11–26

Version 1.1, in comparison to previous versions, significantly increases the internal reasoning capabilities of Leo-III, fixes a lot of minor errors and improves some data structure's performance. As of this version, Leo-III supports all common TPTP [31, 32] dialects (CNF, FOF, TFF, THF) as well as its polymorphic variants [18]. It gives results according to the standardized SZS ontology and, additionally, a TSTP-compatible refutation proof object can be printed if desired. Leo-III version 1.1 will be released to the public at CASC-26 in August 2017.

In this system description, first the theoretical foundations of Leo-III are briefly surveyed. Subsequently, the sequential proof procedure including the external cooperation scheme is presented. Also, implementation details of the underlying data structures and the agent mechanism are presented. Finally, exemplary application scenarios of Leo-III are presented.

2 Polymorphic Higher-Order Logic

Simple type theory, also referred to as classical higher-order logic (HOL), is an expressive logic formalism that allows for higher-order quantification, that is quantification over arbitrary set and function variables. It is based on the simply typed λ-calculus and was originally developed by Church [14]. For thorough discussions of typed λ-calculi we refer to the literature [3]. For Leo-III, a restricted variant of a second-order polymorphic λ-calculus is used that corresponds to HOL with rank-1 polymorphism (simply called HOL in the following).

The set of types $\mathcal{T}$ is thereby generated by the following abstract syntax ($\tau_i \in \mathcal{T}$):

$$\begin{array}{rll} \tau_1, \ldots, \tau_n & ::= \zeta(\tau_1, \ldots, \tau_n) & \text{(Type application)} \\ & \mid \tau_1 \to \tau_2 & \text{(Abstraction type)} \\ & \mid \alpha & \text{(Type variable)} \\ & \mid \forall\alpha.\ \tau_1 & \text{(Polymorphic type)} \end{array}$$

where $\mathcal{S}$ is a non-empty set of sort symbols, $\zeta \in \mathcal{S}$ is a type constructor of arity n and α is a type variable. Type constructors of arity zero are called base types. According to the rank-1 polymorphism restriction, polymorphic types must not occur on the right side of an abstraction type. It may thus, without loss of generality, be assumed that all type abstractions $\forall\alpha.\forall\beta.\ \ldots$ appear as prefix (at prenex position) of a type. We assume that $\mathcal{S}$ consists of at least two elements $\{o, \iota\} \subseteq \mathcal{S}$, both of arity zero, where o and ι denote the type of Booleans and some non-empty domain of individuals, respectively.

The set of HOL terms Λ is then given by the following abstract syntax ($\tau, \nu \in \mathcal{T}$):

$$\begin{array}{rll} s, t ::= X_\tau \in \mathcal{V}_\tau & \mid c_\tau \in \Sigma & \text{(Variable / Constant)} \\ \mid (\lambda x_\tau.\ s_\nu)_{\tau\to\nu} & \mid (s_{\tau\to\nu}\ t_\tau)_\nu & \text{(Abstraction / Application)} \\ \mid (\Lambda\alpha.\ s_\tau)_{\forall\alpha.\ \tau} & \mid (s_{\forall\alpha.\ \tau}\ \nu)_{\tau[\alpha/\nu]} & \text{(Type abstraction / Type application)} \\ \mid (\forall\alpha.\ s_o)_o & & \text{(Type quantification)} \end{array}$$

where c_τ denotes a typed constant from the signature Σ, X_τ is a (term) variable, and α is a type variable. The type of a term is explicitly stated as subscript but may be dropped for legibility reasons if obvious from the context. Note that, for using extrinsic typing, each term $t \in \Lambda$ is inherently well-typed. Terms s_o of type o are called formulae. As a further restriction, only non-polymorphic types are allowed as argument for type applications.

Note that while type quantification $\forall\alpha.s_o$ could easily be defined as $\forall\alpha.s_o \equiv (\Lambda\alpha.\ s_o = \Lambda\alpha.\ T)$ or regarded as a constant application $\Pi^\star_{(\forall\alpha.\ o)\to o}\,(\Lambda\alpha.s_o)$, we need to explicitly include it to the syntax due to the rank-1 polymorphism restriction.

Σ is chosen to contain at least of the primitive logical connectives for disjunction, negation, and (polymorphic) equality, universal quantification and choice. Hence, we have

$\{\vee_{o\to o\to o}, \neg_{o\to o}, =_{\forall\alpha.\,\alpha\to\alpha\to o}, \Pi_{\forall\alpha.\,(\alpha\to o)\to o}, \iota_{\forall\alpha.\,(\alpha\to o)\to\alpha}\} \subseteq \Sigma$. Binder notation is used whenever reasonable, i.e. by convention the term $\Pi_{\forall\alpha.\,(\alpha\to o)\to o}\ \iota\ (\lambda X_\iota.\, s_o)$ is abbreviated by $\forall X_\iota.s_o$. Also, type applications and infix notation are implicit whenever clear from the context, so $s_\tau = t_\tau$ is short for $(=_{\forall\alpha.\,\alpha\to\alpha\to o}\ \tau\ s_\tau\ t_\tau)$. The remaining logical connectives can be defined as usual, e.g. $s \wedge t := \neg(\neg s \vee \neg t)$ or $T := \neg\forall X_o.\, X$.

The semantics of monomorphic HOL can be found in the literature [6, 7, 22]. HOL augmented with quantification over types and type operators was studied in [20]. The variant of HOL presented here is more similar to the one used by HOL2P [33], since it includes universally quantified types (but, as opposed to HOL2P, does not support type operator variables). It is based on a restricted version of $\lambda 2$, also called System-F [15].

As a consequence of Gödel's Incompleteness Theorem, HOL with standard semantics is necessarily incomplete. However, if we assume a generalized notion of HOL semantics, called Henkin semantics [17], a meaningful notion of completeness can be achieved. We assume Henkin semantics in the following.

3 Calculus

The proof search of Leo-III is guided by a refutation-based calculus, i.e. it uses the fact that $A_1, \ldots, A_n \vdash C$ if and only if $S = \{A_1, \ldots, A_n, \neg C\}$ is inconsistent. To that end, the set S consisting of the axioms A_i and the negated conjecture $\neg C$ is transformed into an equisatisfiable set S' of clauses in (equational) clausal normal form. Then, $A_1, \ldots, A_n \vdash C$ is valid, if the empty clause $\square$ can be derived from S', or, equivalently, if $\square \in \overline{S'}$, where $\overline{S'} \supseteq S'$ is a set of clauses closed under the inferences of the calculus.

A method for saturating a given set of HO clauses is resolution [4], as e.g. employed by LEO-II [8]. In first-order theorem proving, superposition [2] – a further restricted form of paramodulation [23] – is probably the most successful calculus, which improves naive resolution not only by an appropriate handling of equality, but also by using ordering constraints to restrict the number of possible inferences. However, so far the generalization of superposition, paramodulation or even ordered resolution to HOL is still in its infancy, let alone the development of appropriate term orderings for higher-order terms.

Leo-III employs a higher-order (ordered) paramodulation calculus with specialized rules for treatment of extensionality, choice and defined equalities.

Higher-Order Paramodulation An equation is a pair $s \simeq t$ of HOL terms, where $\simeq$ is assumed to be symmetric. A literal ℓ is a signed equation, written $[s \simeq t]^\alpha$ where $\alpha \in \{\mathfrak{t}, \mathfrak{f}\}$ represents the polarity of the literal. Non-equality predicates (terms s_o of type o) are represented as literals $[s_o \simeq T]^\alpha$ and may simply be written $[s_o]^\alpha$. A clause $\mathcal{C}$ is a multiset of literals, denoting its disjunction. For brevity, if $\mathcal{C}$ and $\mathcal{D}$ are clauses and ℓ is a literal, we write $\mathcal{C} \vee \ell$ and $\mathcal{C} \vee \mathcal{D}$ for the multi-union $\mathcal{C} \cup \{\ell\}$ and $\mathcal{C} \cup \mathcal{D}$, respectively.

The higher-order paramodulation calculus $\mathcal{EP}$ is given by the union $\mathcal{EP} = \mathcal{INF} \cup \mathcal{EXT} \cup \mathcal{UNI} \cup \mathcal{CNF}$ of the primary inference rules $\mathcal{INF}$, extensionality rules $\mathcal{EXT}$, unification rules $\mathcal{UNI}$ and clause normal form rules $\mathcal{CNF}$. For simplicity, the rules displayed here are variants of the actual calculus rules for monomorphic HOL.

The primary inference rules $\mathcal{INF}$ are given by

$$\frac{\mathcal{C} \vee [l \simeq r]^{\mathfrak{t}} \qquad \mathcal{D} \vee [s \simeq t]^\alpha}{\mathcal{C} \vee \mathcal{D} \vee [s[r]_\pi \simeq t]^\alpha \vee [s|_\pi \simeq l]^{\mathfrak{f}}}\ (\text{Para}) \qquad \frac{\mathcal{C} \vee [l \simeq r]^\alpha \vee [s \simeq t]^\alpha}{\mathcal{C} \vee [l \simeq r]^\alpha \vee [l \simeq s]^{\mathfrak{f}} \vee [r \simeq t]^{\mathfrak{f}}}\ (\text{EqFac})$$

$$\frac{\mathcal{C} \vee [X_{\overline{\tau_i} \to o}\ \overline{t^i_{\tau_i}}]^\alpha \qquad P \in \mathcal{AB}^c_{\overline{\tau_i} \to o}}{\big(\mathcal{C} \vee [X_{\overline{\tau_i} \to o}\ \overline{t^i_{\tau_i}}]^\alpha\big)\{X/P\}}\ (\text{PrimSubst})$$

where $s|_\pi$ is the subterm of s at position π, and $s[r]_\pi$ denotes the term that is created by replacing the subterm of s at position π by r. Since unification in HOL is undecidable, the unification tasks are encoded as negative equality literals – as seen in (Para) and (EqFac) – which may again be subject of further inferences. $\mathcal{UNI}$ defines further treatment of such unification literals. Intuitively, paramodulation is a conditional rewriting step that is justified if the unification tasks can be solved.

In HOL, the additional rule (PrimSubst) is required for completeness. Primitive substitutions guess the top-level logical structure of the instantiated term, while further decisions on P are delayed. The instantiations $\mathcal{AB}^c_{\overline{\tau_i} \to o}$ are called approximating bindings for type $\overline{\tau_i} \to o$ and head symbol c.

The above rule (Para) is unordered and produces (in particular, in our higher-order setting) numerous irrelevant and redundant clauses. In order to restrict inference rules such as (Para), Leo-III employs a higher-order term ordering, called *computability path ordering* [10], which has been investigated primarily for termination proofs. While the development of a complete calculus is still ongoing research, first results of these ordering constraints seem promising.

Further rules The inference rules $\mathcal{CNF}$ for clause normalization are omitted here. The unification rules $\mathcal{UNI}$ are a variant of Huet's pre-unification augmented with type unification rules (cf. [4]). Extensionality aspects of HOL with Henkin semantics are dealt with on the calculus level rather than postulating corresponding extensionality axioms. The rules $\mathcal{EXT}$ are analogous to those of LEO-II [8].

Furthermore, Leo-III adapts specialized inference rules for replacing defined (Leibniz and Andrews) equalities and handling choice from its predecessor [9]. Additionally, exhaustive instantiations of universally quantified variables of finite types prior to CNF calculation is supported.

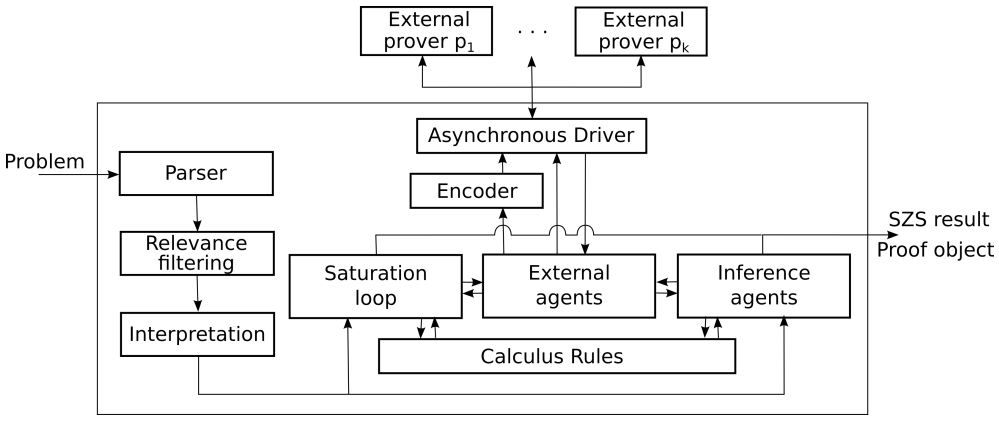

Figure 1: Functional design of Leo-III's refutation process

4 Proof Procedure

Due to the agent-based architecture of Leo-III, the overall proof procedure is slightly more complex compared to its predecessor. Leo-III's functional design is schematically displayed in Fig. 1. An invocation of the prover proceeds as follows:

(i) Reading the problem file locally or via a network connection to a specified URL

(ii) Parsing of the input stream to an internal abstract syntax tree (AST) representation

(iii) Axiom pruning using a relevance filter (if a conjecture is present)

(iv) Interpretation, type checking and translation of the AST to an equivalent λ-term

(v) The resulting terms are given to the main refutation procedure

  (a) An agent is invoked which implements a traditional saturation loop (see below)

  (b) In parallel, multiple independent agents (cf. §5.3) cooperatively try to find a proof

  (c) At any point, both components may asynchronously invoke external provers

(vi) If a contradiction is found, a proof is reconstructed by recursive backward traversal of the empty clause's inference parents

(vii) An SZS [30] output is returned together with the refutation proof (if desired)

A dedicated internal reasoning agent (cf. (v)-(a) above) implements a stand-alone reasoning loop inspired by the given-clause algorithm of E [27]. A simplified version is given by

```
 1  U := preprocess(input)
 2  P := ∅
 3  while(U ≠ ∅)
 4    checkExternal()
 5    g := selectBest(U)
 6    g := simp(g, P)
 7    g := ext(g)
 8    if (¬clausalNormal(g)) U := U ∪ cnf(g)
 9    else if (definesChoiceSymbol(g)) registerChoiceSymbol(g)
10    else
11      if (g = □) Theorem
12      else if (¬redundant(g, P))
13        submitExternal(P)
14        P := P \ {p ∈ P | redundant(p, {g})}
15        P := P ∪ {g}
16        T := generate(g, P)
17        T := T ∪ findChoice(g, P)
18        T := simp(cnf(unify(T)))
19        U := U ∪ {t ∈ T | ¬trivial(t)}
20      endif
21    endif
22  end
```

The algorithm structures the search space using two sets U and P of *unprocessed* clauses and *processed* clauses, respectively, where initially all input clauses are considered unprocessed. Intuitively, the algorithm iteratively selects an unprocessed clause g (the *given clause* [27])

from U and inserts all inferences between g and clauses in P as fresh clauses into U until either the empty clause is found or there are no unprocessed clauses left[1].

To that end, the input problem is first pre-processed (including expansion of defined symbols, simplification, miniscoping, CNF transformation and further) and added to U. Clause selection (line 5) is inspired by E's usage of multiple selection heuristics aligned in a weighted round-robin scheme (see §5.1). The simplification methods `simp` (lines 6,18), in the current version, do not yet include rewrite simplifications. This is a feature planned for the next release. The `ext` method (line 7) applies (FuncExt) to the given clause g, if applicable. Since this may create non-CNF formulas, we reinsert clauses that require CNF normalization (line 8) to the unprocessed set U. `redundant`(c, S) checks for subsumption of c by a set of clauses S. This method is used for forward subsumption (line 12) as well as for backward subsumption (line 14). The application of generating inferences (Para), (EqFac), (PrimSubst), (BoolExt) as well as choice and defined equality rules are denoted `generate`, where at least one premise to each rule must be g. Since `generate` may also create non-CNF formulas, normalization is applied afterwards i.e. after eagerly solving unification constraints (line 18). Finally, only non-`trivial` generated clauses are added to U (line 19), i.e. clauses that are not tautological.

Choice handling is displayed in lines 9 and 17, where the first operation removes clauses representing the axiom of choice (AC) for a specific symbol and the latter operation inserts concrete AC instances (cf. [9]).

Finally, external prover cooperation is displayed in lines 4 and 13: `checkExternal` checks for external prover results in a non-blocking fashion and returns any helpful result, if existent. If a helpful answer is returned, the saturation loop terminates and returns the external result instead. `submitExternal` requests a external prover invocation with respect to the current set of processed clauses P. The number of parallel open request per external reasoning systems can be limited (two by default).

5 Implementation

Leo-III is implemented in Scala[2] based on the associated system platform LEOPARD [36]. The latter provides a reusable framework and infrastructure for higher-order deduction systems, consisting of fundamental generic data structures for typed λ-terms, indexing means, a generic agent-based blackboard architecture, parser, and proof printer. Leo-III makes heavy use of these data structures and support means, and, on top of them, implements its specific calculus rules, control heuristics, the proof procedure, specialist agents and further functionality. Fig. 2 provides an overview on the components of Leo-III (roughly corresponding to Scala packages) and their utilization relations indicated by arrows. Components surrounded by dotted lines are provided (at least in part) by LEOPARD. The generic data structures, logical data structures and indexing data structures are used by almost every other component so we omit the usage relation arrows here. The calculus component provides the implementation of the inference rules described in section 3. This package is used by the components *Control* and *Agents* which both abstract from inference rules and represent a layer of (heuristic) calculus rule applications relative to the current prover state. A major component, the *External* package, provides abstractions for external reasoning systems as well as generic means for asynchronous communication. Finally, the *Encoding* component provides various first-order transformation algorithms

[1] Since the set of clauses to be processed grows without bounds in general, a time limit is given to the system. If the empty clause is not found within this limit, a timeout result (SZS_Timeout) is returned.

[2] Leo-III uses Scala 2.11.8 and requires Java 1.8. Previous versions of underlying Java distributions are not supported due to limitations in system process management.

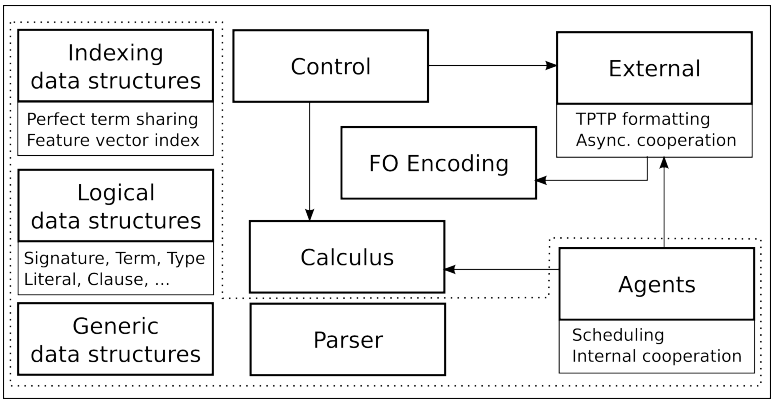

Figure 2: Overview of Leo-III's components.

as well as pretty printer for different TPTP languages.

During the development of Leo-III, careful attention has been paid to providing maximal compatibility with existing systems and conventions of the peer community, especially to those of the TPTP infrastructure [31]. Leo-III reads problems in every standard dialect of the TPTP syntax, including the higher-order THF dialect [32] and the (untyped and typed) first-order dialects[3] FOF and TFF [31]. New to version 1.1 of Leo-III is the support for the recent polymorphic variants TF1 and TH1 [18] of typed first-order and higher-order languages, respectively. Results are printed out in TSTP format using SZS ontology [30] values. Additionally, Leo-III can give a TSTP compatible proof object, e.g. for proof verification by GDV [31] or proof reconstruction in the context of Isabelle's Sledgehammer tool [24].

5.1 Data Structures

Data structure choices are a critical part of a theorem prover and permit reliable increases of overall performance when implemented and exploited properly. Key aspects for efficient theorem proving have been an intensive research topic and have reached maturity within FO ATPs. In the context of the Leo-III prover, quite some effort was invested into designing appropriate data structures for HOL reasoning procedures. While their development is still far from being as mature and optimized as their first-order counterparts, the current data structures seem to yield good results in practice. Two of the more interesting structure's implementation details are surveyed in the following.

5.1.1 Representation of λ-Terms

While currying is – from a theoretical point of view – an elegant technique for a uniform treatment of functions of all arities, it comes with a major drawback for automation. There are many logical procedures where the head symbol of a term needs to be accessed or the individual arguments of an application need to be examined in the left-to-right reading order (e.g. unification or matching procedures). However, in a naive curried representation of a λ-term the head symbol of a term may be deeply buried under several layers of applications.

[3] First-order problems are internally translated to higher-order terms and, in the current version, treated as regular higher-order problems.

Hence for each head access a linear number of traversal operations need to be performed. A similar phenomenon applies for left-to-right argument traversal.

To overcome this weakness of a classical term representation, Leo-III uses a so-called spine notation [12], which imitates first-order-like terms in a higher-order setting. Here, terms are either type abstractions, term abstractions or applications of the form $f \cdot (s_1; s_2; \ldots)$ where the head f is either a constant symbol, a bound variable or a complex term and the spine $(s_1; s_2; \ldots)$ is a linear list of arguments that are, again, spine terms. Note that if a term is β-normal, f cannot be a complex term. This observation led to a internal implementation distinction between β-normal and (possibly) non-β-normal spine terms where the first kind has an optimized representation whose head having only associated a reference to an integer representing the constant symbol or variable (cf. further below).

Additionally, the term representation employs explicit substitutions [1]. In a setting of explicit substitutions, substitutions are part of the term language and can thus be postponed and composed before being applied to the term. This technique admits more efficient β-normalization resp. substitution operations as terms are only traversed once, regardless of the number of substitutions applied.

Nameless Representation The term data structure uses a locally nameless representation both at the type and term level, that extends de-Bruijn indices to (bound) type variables [19]. The definition of de-Bruijn indices [11] for type variables is analogous to the one for term variables. One of the most important advantages of nameless representations over representations with explicit variable names is that α-equivalence is reduced to syntactical equality, i.e. two terms are α-equivalent if their nameless representation is equal. Together with the term indexing of Leo-III (cf. §5.1.2) this yields constant time operations for checking α-equivalence (analogously for types).

An example term in an abstract representation of the here described data structure reads

$$\Lambda.\lambda_{\underline{1}\to o}.f_{\forall(\underline{1}\to o)\to o\to o} \cdot \left(\underline{1}; \lambda_{\underline{1}}.\ 2 \cdot 1; c_o\right)$$

where $f, c \in \Sigma$ and the de-Bruijn indices for type variables are underlined.

5.1.2 Indexing Structures

Indexing data structures are popular in first-order theorem proving for speeding up querying of terms, literals or clauses wrt. certain conditions (the indexing relation). Employment of indexing methods improves operations that are frequently invoked during the proof procedure, such as finding unifiable terms or clauses for subsumption.

In higher-order theorem proving, there exist only a few indexing approaches. This is due to the fact that most operations for building a term index are undecidable, e.g., computing the most specific generalization, or higher-order unification. For the latter case, however, there exists a decidable unification fragment, so called higher-order pattern unification [21], but algorithms for those fragment are highly complex and seem not to be efficient in practice [25, 26].

Term Sharing Terms are perfectly shared within Leo-III, meaning that each term is only constructed once and then reused between different occurrences. This not only reduces memory consumption in large knowledge bases, but also allows constant-time term comparison for syntactic equality using the term's pointer to its unique physical representation. For fast (sub-)term retrieval based on syntactical criteria (e.g. head symbol, subterm occurrences, etc.) from the term indexing mechanism, terms are kept in β-normal η-long form. Leo-III comes with

a number of different (heuristic) β-normalization strategies that adjust the standard leftmost-outermost strategy with different combinations of strict and lazy substitution composition resp. normalization and closure construction [29].

Subsumption Indexing In Leo-III a higher-order adaption of feature vector indexing [28] is employed. It is used to reduce the number of subsumption tests during the proof procedure. The adapted feature vector index has limitations when indexing terms with variables at head positions but nevertheless seems suitable in practice. A formal, more thorough investigation about its benefits in this setting is further work.

5.2 External cooperation

As pointed out before, Leo-III's agents may at any point invoke external reasoning tools. To that end, Leo-III includes an encoding module which translates (polymorphic) higher-order clauses to polymorphic and monomorphic typed first-order clauses. While LEO-II relied on cooperation with untyped first-order provers, we hope to reduce clutter and therefore achieve better results using native type support in first-order provers. Further cooperation with other TPTP-compliant FO or HO reasoning tools is supported, e.g. for using higher-order counter model finder. To that end, Leo-III supports output in TF0, TF1, TH0 and TH1 syntax.

5.3 Agents

An *agent* is a software component that can be executed independently from others. Moreover, an agent is given the ability to decide on its own when to execute its functionality. This high amount of autonomy is a key feature of agents [34]. In the Leo-III system, agents are employed as *specialists* for some aspects of the proof search. The underlying architecture of Leo-III employs a blackboard data structure which agents collaboratively use for finding a proof. The work of the agents is thereby divided in transactional tasks and organized as auctions, in which it is decided what tasks to be executed next in case of interference [35, 36]. In Leo-III agents can be utilized in two different ways: First, top down, where one agent implements a sequential loop and all remaining agents perform subsidiary or computationally heavy tasks in parallel. Second, bottom up, as a set of inference agents where each agent is in charge for the application of a single inference rule.

The underlying architecture consists of three main components: A blackboard data structure containing the shared proof state, the agents and their execution, and the scheduler that coordinates an agent's access to the blackboard. Figure 3 displays the overall interaction between these components. In the following the components of the architecture are briefly described, as well as two example agent implementations are given.

Blackboard Architecture. In a blackboard architecture communication focuses on the globally shared data [34]. Agents do not communicate directly with other agents, but do so indirectly by manipulating the data in the blackboard. The Leo-III blackboard [36] is a collection of globally shared and accessible data structures any agent can query and manipulate at any time in parallel. The set of data structures itself is not fixed, as any object implementing the `DataStore` interface can be added, even during execution. Hence, an agent will not insert data into data structures itself, but will pass the proposed insertion to the blackboard. The blackboards main purpose is the organization and presentation of all shared data.

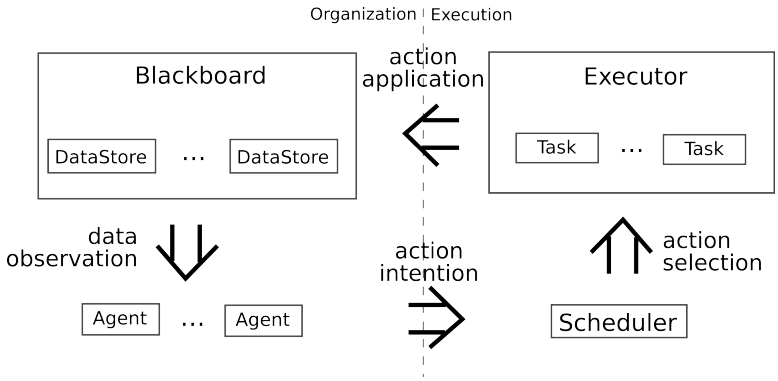

Figure 3: Information- and work flow of the Blackboard-Agent-Interaction.

Agents. Whereas the blackboard organizes and presents the data in the system, agents are specialized, autonomous components manipulating this data. Classically, agents are composed of three components: environment perception, decision making, and action execution [34].

Environment perception is handled by both a subscriber pattern that triggers on changes within relevant part of the blackboard and the provision of data through its maintained data structures. The two remaining components are separated in the here employed agent abstraction. The first part, the interface `Agent`, is responsible for the decision making. Upon the triggering of changed or new data, a function `filter` is called. This function captures the intention of the agent and returns a sequence of functionality invocations an agent wants to execute. The intended work of multiple agents might interfere, e.g. one agent removes a redundant clause and another wishes to perform a paramodulation with the same clause. To that end, agent's actions are abstracted by the interface `Task`. A `Task` capsules a transaction which results in a so-called Delta, containing the changes to be applied to the blackboard. A `Task` might not be executed if it is blocked by an interfering task.

Auction Scheduler. Since computational resources are limited, only a selection of the generated tasks can be executed. The selection addresses two important aspects: Firstly, a ranking that prefers the execution of most promising tasks. Secondly, a collision detection such that only sets of non interfering tasks are executed. The implementation for this selection is given by an approximation algorithm for combinatorical auctions [13]. More precisely, each agent computes and places a bid for the execution of its task(s). The scheduler then tries to maximize the global benefit of the particular set of actions to choose. In the current version, the agents assign value to their tasks purely heuristic. The calculation of task importance based on machine learning and the proof state is ongoing development.

Monolithic and Subsidiary Agents One important mode in which Leo-III can utilize the agents is the so-called monolithic mode. In this mode, a interleavable sequential loop is executed by one agent. Since agents actions are executed in transactions (tasks), the sequential loop represents a single loop iteration as a dedicated task. In parallel to this loop, subsidiary agents can be employed performing supportive (maybe non-essential) tasks and handle computationally expensive, parallelizable tasks. An exemplary subsidiary agent is a higher-order pre-unification agent, that solves all unification constraints generated during one loop iteration in parallel.

Other subsidiary agents implemented in Leo-III are subsumption agents that remove redundant clauses and agents invoking external provers.

Rule Agents and Compound Rule Creation From an opposite point of view, we can explore the implementation of an ATP system by implementing its calculus rules directly as independent agents. In this context, one agent is provided for each inference rule. Here, an agent observes the blackboard for possible candidate formulas. If its associated rule is applicable, a task is created whose result will insert the new formulas (or update existing ones). Exploiting the `Rule` interface, further compound rules can be constructed, that is, combinations of inference rules that should always be executed directly after each other.

Since the vast amount of tasks generated by this approach is, at the moment, guided only by heuristic selection, the approach is not competitive and still experimental. Ongoing work includes guidance of rule agents with machine learning methods. Additionally, the automatic generation of compound rules based on machine learning from previous proof experience is current research.

5.4 Further Components

The underlying LeoPARD framework provides useful stand-alone components. For example, a generic parser is provided that supports all TPTP syntax dialects. It is implemented using ANTLR4 and converts its produced concrete syntax tree to an internal TPTP AST data structure which can be used to implement further stand-alone procedures. A pretty printer which converts Leo-III's internal term representation to THF and TFF format is used for external cooperation.

6 Applications and Examples

Apart its the theorem proving capabilities, Leo-III supports additional features for its input. Given the flag `--consistency`, the problems axioms are checked for consistency, depending on the overall mode in advance or in parallel to the proof search. Calling the program with `--toTHF`, the parser can be employed to convert any TPTP problem into THF. If the conversion is not needed a call with `--syntaxCheck` checks the problem for correct TPTP syntax and the flag `--typeCheck` runs the internal type checker on the parsed problem to verify any correct input up to TH1. The verbose proof object can be reduced with the option `--proofcompression`. This mode skips less informative non-branching steps in the proof and compress them into a sequence of inference applications (cf. section 6.3). To this flag a list of inference rule names can be passed to customize the compression.

6.1 Leo-III as Meta Prover

The architecture of Leo-III allows for an easy employment as a meta prover. As described in section 5.2 Leo-III, is already able to integrate external reasoners into its own proof procedure. Combining these external agents and some preprocessing and translation options, Leo-III can be run purely as a meta system coordinating provers and translating between them, similar to Isabelle's Sledgehammer [24].

As a meta prover, Leo-III provides axiom selection, pre-processing techniques, encoding into all TPTP formats, as well as asynchronous, parallel external prover support. In this mode, different preprocessing settings and axiom selections can be scheduled to the different provers.

While this meta prover mode of Leo-III is not yet optimized towards high performance, all required components are available an can already be used to create ad-hoc solutions.

6.2 Reasoning in Non-Classical Logics

One major goal of Leo-III is to provide native means of reasoning within (and about) non-classical logics including free logic, quantified conditional logic, and quantified modal logic. The reasoning in such non-classical logics is enabled by a semantical embedding of the target logic into HOL. Detailed information about this approach can be found in [5] (see also the references therein). Such logics are of strong interest in many different fields of research, for example in mathematics, artificial intelligence, and philosophy. In its current state, our system is already capable of reasoning for a range of embedded logics using an external pre-processor [16]. The next version of Leo-III will natively integrate such a pre-processor in order to offer out-of-the-box automation for non-classical logics.

6.3 Example

A prominent example easily expressible in HOL is the surjective Cantor theorem. It can be stated as:

```
thf(sur_cantor, conjecture, (~ ( ? [F: $i > ($i > $o)] : (
                                  ! [Y: $i > $o] :
                                   ? [X: $i] : (
                                     (F @ X) = Y
                                   )
                                ) )),file('sur_cantor.p',sur_cantor)).
```

Leo-II gives verifiable proofs. Interesting steps from a "proof idea" point of view are formulae 41 and 95; in these steps a diagonalization argument is constructed by (PrimSubst) and unification, respectively. A compressed version of the proof output (via `--proofcompression` flag) is displayed below. Unfortunately, compressed proofs cannot, at the moment, be processed by GDV. However, the uncompressed variant was verified by GDV in 351 seconds (see GDV output at Appendix A).

```
% SZS status Theorem for sur_cantor.p
% SZS output start CNFRefutation for sur_cantor.p
thf(sk5_type,type,(sk5: $i > $i > $o )).
thf(sk6_type,type,(sk6: ( $i > $o ) > $i )).
...
thf(2,negated_conjecture,(
    ~ ( ~ ( ? [A: ( $i > $i > $o )] :
              ! [B: ( $i > $o )] :
              ? [C: $i] :
                ( ( A @ C )
                = B ) ) ) ),
    inference(neg_conjecture,[status(cth)],[sur_cantor])).
thf(17,plain,(
    ! [B: $i,A: ( $i > $o )] :
      ( ( sk5
        @ ( sk6
          @ ^ [C: $i] :
              ( A @ C ) )
        @ B )
      | ~ ( A @ B ) ) ),
    inference(bool_ext,[status(thm)],
      inference(func_ext,[status(esa)],
        inference(lifteq,[status(thm)],
          inference(cnf,[status(esa)],
            inference(polarity_switch,[status(thm)],
              inference(defexp_and_simp_and_etaexpand,[status(thm)],[2])))))))).
thf(41,plain,(
```

```
      ! [B: ( $i > $o ),A: $i] :
        ( ( sk5
          @ ( sk6
            @ ^ [C: $i] :
                ~ ( B @ C ) )
          @ A )
        | ( B @ A ) ) ),
      inference(simp,[status(thm)],
        inference(cnf,[status(esa)],
          inference(prim_subst,[status(thm)],
          [17:[bind(A,$thf(^ [D: $i] : ~ ( C @ D )))]])))).
thf(95,plain,
      ( sk5
      @ ( sk6
        @ ^ [A: $i] :
            ~ ( sk5 @ A @ A ) )
      @ ( sk6
        @ ^ [A: $i] :
            ~ ( sk5 @ A @ A ) ) ),
      inference(pre_uni,[status(thm)],
        inference(eqfactor_ordered,[status(thm)],[41])
        :[bind(A,$thf(sk6 @ ^ [C: $i] : ~ ( sk5 @ C @ C ))),
          bind(B,$thf(^ [C: $i] : ( sk5 @ C @ C )))])).
...
thf(141,plain,(
      $false ),
      inference(pattern_uni,[status(thm)],
        inference(paramod_ordered,[status(thm)],[95,108]))).
% SZS output end CNFRefutation for sur_cantor.p
```

Deduction of formula 2 is simple negation of the input conjecture. Formula 17 is inferred by CNF calculation and application of extensionality rules. Formula 41 is generated by rule (PrimSubst) from formula 17 via instantiation of the universally quantified variable A by an approximating binding for $\neg$ and subsequent simplification, i.e. instantiation with the term $\lambda \mathtt{D}_\iota.\, \neg(\mathtt{C}\ \mathtt{D})$, where C is a fresh variable to the clause. Subsequently, in formula 95 the variable A is instantiated by a set that does not contain those elements contained in the image (the power set) of the original function f. An analogous inference counter part branch (yielding in formula 108) is not displayed due to space limitations. After paramodulation (and unification) both these branches yield the empty clause (formula 141) and hence a proof for the initial conjecture.

7 Summary and Further Work

Leo-III is a cooperative higher-order theorem prover, which extends and further improves on the ideas underlying its predecessor system LEO-II. Unlike LEO-II, Leo-III implements a higher-order (ordered) paramodulation calculus that is employed within a given-clause saturation procedure and orchestrated (in different forms) within an agent-based architecture for parallel proof search. Cooperation with first- and higher-order reasoning systems such as theorem provers or model finders is supported.

Future work includes more experimentation with external provers and their flag settings, as well as internal parameter optimization for clause selection and further aspects of the proof search. Moreover, a thorough evaluation of the agent-based reasoning approach needs to be performed in order to optimally adjust the architecture for various platforms and applications.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991.

[2] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

[3] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

[4] 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.

[5] Christoph Benzmüller. Invited talk: On a (quite) universal theorem proving approach and its application in metaphysics. In Hans De Nivelle, editor, *TABLEAUX 2015*, volume 9323 of *LNAI*, pages 213–220, Wroclaw, Poland, 2015. Springer. (Invited paper).

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

[7] Christoph Benzmüller and Dale Miller. Automation of higher-order logic. In Jörg Siekmann, Dov Gabbay, and John Woods, editors, *Handbook of the History of Logic, Volume 9 — Logic and Computation*. Elsevier, 2014. In print.

[8] Christoph Benzmüller, Lawrence C. Paulson, Nik Sultana, and Frank Theiß. The higher-order prover LEO-II. *Journal of Automated Reasoning*, 55(4):389–404, 2015.

[9] Christoph Benzmüller and Nik Sultana. LEO-II version 1.5. In Jasmin Christian Blanchette and Josef Urban, editors, *PxTP 2013*, volume 14 of *EPiC Series*, pages 2–10. EasyChair, 2013.

[10] Frédéric Blanqui, Jean-Pierre Jouannaud, and Albert Rubio. The computability path ordering. *Logical Methods in Computer Science*, 11(4), 2015.

[11] N.G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *INDAG. MATH*, 34:381–392, 1972.

[12] Iliano Cervesato and Frank Pfenning. A linear spine calculus. *J. Log. Comput.*, 13(5):639–688, 2003.

[13] Georgios Chalkiadakis, Edith Elkind, and Michael Wooldridge. *Computational Aspects of Cooperative Game Theory*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2011.

[14] Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[15] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[16] Tobias Gleiner, Alexander Steen, and Christoph Benzmüller. Theorem provers for every normal modal logic. In Thomas Eiter and David Sands, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, EPiC Series in Computing, Maun, Botswana, 2017. EasyChair. To appear.

[17] Leonard Henkin. Completeness in the theory of types. *J. Symb. Log.*, 15(2):81–91, 1950.

[18] Cezary Kaliszyk, Geoff Sutcliffe, and Florian Rabe. TH1: the TPTP typed higher-order form with rank-1 polymorphism. In Pascal Fontaine, Stephan Schulz, and Josef Urban, editors, *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning, Coimbra, Portugal.*, volume 1635 of *CEUR Workshop Proceedings*, pages 41–55. CEUR-WS.org, 2016.

[19] A.J. Kfoury, S. Ronchi della Rocca, J. Tiuryn, and P. Urzyczyn. Alpha-conversion and typability. *Information and Computation*, 150(1):1 – 21, 1999.

[20] Thomas F. Melham. The HOL logic extended with quantification over type variables. *Formal Methods in System Design*, 3(1-2):7–24, 1993.

[21] Dale Miller. Unification of simply typed lambda-terms as logic programming. In *In Eighth International Logic Programming Conference*, pages 255–269. MIT Press, 1991.

[22] Reinhard Muskens. Intensional models for the theory of types. *Journal of Symbolic Logic*, pages 98–118, 2007.

[23] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 371–443. Elsevier and MIT Press, 2001.

[24] Tobias Nipkow, Lawrence C. Paulson, and Makarius Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science. Springer, 2002.

[25] Brigitte Pientka. Higher-order term indexing using substitution trees. *ACM Trans. Comput. Logic*, 11(1):6:1–6:40, November 2009.

[26] Brigitte Pientka and Frank Pfenning. Optimizing higher-order pattern unification. In *19th International Conference on Automated Deduction*, pages 473–487. Springer-Verlag, 2003.

[27] Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2,3):111–126, August 2002.

[28] Stephan Schulz. Simple and efficient clause subsumption with feature vector indexing. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788 of *LNCS*, pages 45–67. Springer, 2013.

[29] Alexander Steen and Christoph Benzmüller. There Is No Best β-Normalization Strategy for Higher-Order Reasoners. In M. Davis, A. Fehnker, A. McIver, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 9450 of *LNAI*, pages 329–339, Suva, Fiji, 2015. Springer.

[30] Geoff Sutcliffe. The SZS Ontologies for Automated Reasoning Software. In *LPAR Workshops*, volume 418, 2008.

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

[32] Geoff Sutcliffe and Christoph Benzmüller. Automated reasoning in higher-order logic using the TPTP THF infrastructure. *J. Formalized Reasoning*, 3(1):1–27, 2010.

[33] Norbert Völker. HOL2P – a system of classical higher order logic with second order polymorphism. In *Theorem Proving in Higher Order Logics*, pages 334–351. Springer, 2007.

[34] Gerhard Weiss, editor. *Multiagent Systems*. MIT Press, 2013.

[35] Max Wisniewski and Christoph Benzmüller. Is it reasonable to employ agents in theorem proving? In Jan van den Heerik and Joaquim Filipe, editors, *Proceedings of the 8th International Conference on Agents and Artificial Intelligence*, volume 1, pages 281–286, Rome, Italy, 2016. SCITEPRESS – Science and Technology Publications, Lda.

[36] Max Wisniewski, Alexander Steen, and Christoph Benzmüller. Leopard - A generic platform for the implementation of higher-order reasoners. In Manfred Kerber et al., editors, *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, volume 9150 of *LNCS*, pages 325–330. Springer, 2015.

A GDV output for **sur_cantor**

```
% START OF SYSTEM OUTPUT
SUCCESS: Derivation has unique formula names
SUCCESS: All derived formulae have parents and inference information
SUCCESS: Derivation is acyclic
SUCCESS: Assumptions are propagated
SUCCESS: Assumptions are discharged
SUCCESS: Leaf axioms are satisfiable
 RESULT: 2.thm.dis - Isabelle---2016 says Theorem - CPU = 47.75
SUCCESS: 2 is a thm of 1  (Negated cth)
[...]
 RESULT: 198.thm.dis - Isabelle---2016 says Theorem - CPU = 48.48
SUCCESS: 198 is a thm of 112 192
 RESULT: 205.thm.dis - Isabelle---2016 says Theorem - CPU = 48.51
SUCCESS: 205 is a thm of 198
SUCCESS: Derived formulae are verified
SUCCESS: Verified
SZS status Verified
[...]
% RESULT: SOT_oJ7tTg - GDV---0.0 says Verified - CPU = 1329.90 WC = 351.47  CPUTime = 1224.91
% OUTPUT: SOT_oJ7tTg - GDV---0.0 says Verification - CPU = 1329.90 WC = 351.47
```