LeoPARD — A Generic Platform for the Implementation of Higher-Order Reasoners*

Max Wisniewski, Alexander Steen and Christoph Benzmüller

Dept. of Mathematics and Computer Science, Freie Universität Berlin, Germany
max.wisniewski|a.steen|c.benzmueller@fu-berlin.de

Abstract. LeoPARD supports the implementation of knowledge representation and reasoning tools for higher-order logic(s). It combines a sophisticated data structure layer (polymorphically typed λ-calculus with nameless spine notation, explicit substitutions and perfect term sharing) with an ambitious multi-agent blackboard architecture (supporting prover parallelism at the term, clause and search level). Further features of LeoPARD include a parser for all TPTP dialects, a command line interpreter and generic means for the integration of external reasoners.

1 Introduction

LeoPARD (**L**eo's **P**arallel **AR**chitecture and **D**atastructures) is designed as a generic system platform for implementing higher-order (HO) logic based knowledge representation and reasoning tools. In particular, LeoPARD provides the base layer of the new HO automated theorem prover (ATP) Leo-III, the successor of the well known provers LEO-I [3] and LEO-II [6].

Previous experiments with LEO-I and the OAnts mechanism [4] provided good evidence for a flexible, multi-agent blackboard architecture for automating HO logic [5]. However, (due to project constraints) such an approach has not been realized in LEO-II. Instead, the focus has been on the proof search layer in combination with a simple, sequential collaboration with an external first-order (FO) ATP. LEO-II also provides improved term data structures and term indexing and term sharing mechanisms, which unfortunately have not been extensively exploited at the clause and the proof search layer.

For the development of Leo-III the philosophy therefore has been to allocate sufficient resources for the initial development of a flexible and reusable system platform. The goal has been to bundle, improve and extend the most prospective features of the predecessor systems LEO-I, LEO-II and OAnts.

The result of this initiative is LeoPARD[1], which is written in Scala and currently consists of approx. 13000 lines of code. LeoPARD combines a sophisticated data structure layer [17] (polymorphically typed λ-calculus with nameless spine notation, explicit substitutions and perfect term sharing), with a multi-agent blackboard architecture [21] (supporting prover parallelism at the term,

* This work has been supported by the DFG under grant BE 2501/11-1 (Leo-III).
[1] LeoPARD can be download at: https://github.com/cbenzmueller/LeoPARD.git.

clause and search level) and further tools including a parser for all TPTP [18, 19] syntax dialects, generic support for interfacing with external reasoners, and a command line interpreter. Such a combination of features and support tools is, up to the authors knowledge, not matched in related HO reasoning frameworks.

The intended users of the LEOPARD package are implementors of HO knowledge representation and reasoning systems, including novel ATPs and model finders. In addition, we advocate the system as a platform for the integration and coordination of heterogeneous (external) reasoning tools.

2 Term Data Structure

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 [15, 16]. Naturally, one would expect an even higher impact of the data structure choices in HO-ATPs. However, in the latter context, comparably little effort has been invested yet – probably also because of the inherently more complex nature of HO logic.

Term Language. The LEOPARD term language extends the simply typed λ-calculus with parametric polymorphism, yielding the second-order polymorphically typed λ-calculus (corresponding to $\lambda 2$ in Barendregt's λ-cube). In particular, the system under consideration was independently developed by Reynolds [14] and Girard [12] and is commonly called System F today. Further extensions, for example to admit dependent types, are future work.

Thus, LEOPARD supports the following type and term language:

$$\begin{array}{lll} \tau, \nu ::= t \in T & \text{(Base type)} & \\ \quad \mid \alpha & \text{(Type variable)} & s, t ::= X_\tau \in \mathcal{V}_\tau \mid c_\tau \in \Sigma \quad \text{(Variable/Constant)} \\ \quad \mid \tau \to \nu & \text{(Abstraction type)} & \quad \mid (\lambda x_\tau\ s_\nu)_{\tau\to\nu} \mid (s_{\tau\to\nu}\ t_\tau)_\nu \quad \text{(Term abstr./appl.)} \\ \quad \mid \forall \alpha.\ \tau & \text{(Polymorphic type)} & \quad \mid (\Lambda\alpha\ s_\tau)_{\forall\alpha\ \tau} \mid (s_{\forall\alpha\ \tau}\ \nu)_{\tau[\alpha/\nu]} \quad \text{(Type abstr./appl.)} \end{array}$$

An example term of this language is: $\Lambda\alpha\lambda P_{\alpha\to o}\ ((f_{\forall\beta\ (\beta\to o)\to o\to o}\ \alpha)\ (\lambda Y_\alpha\ P\ Y))\ T_o$.

Nameless Representation. Internally, LEOPARD employs a locally nameless representation (both at the type and term level), that extends de-Bruijn indices to (bound) type variables [13]. The definition of de-Bruijn indices [10] for type variables is analogous to the one for term variables. Thus, the above example term is represented namelessly as $\left(\Lambda\lambda_{\underline{1}\to o}\ ((f_{\forall(\underline{1}\to o)\to o\to o}\ \underline{1})\ (\lambda_{\underline{1}}\ 2\ 1))\ T_o\right)$, where de-Bruijn indices for type variables are underlined.

Spine Notation and Explicit Substitutions. On top of nameless terms, LEOPARD employs spine notation [11] and explicit substitutions [1]. The first technique allows quick head symbol queries and efficient left-to-right traversal, e.g. for unification algorithms. The latter augments the calculus with substitution closures that admit efficient (partial) β-normalization runs. Internally, the above example reads $\Lambda\lambda_{\underline{1}\to o}\ f_{\forall(\underline{1}\to o)\to o\to o} \cdot (\underline{1}; \lambda_{\underline{1}}\ 2 \cdot (1); T)$, where $\cdot$ combines function *heads* to argument lists (*spines*) in which ; denotes concatenation of arguments.

Term Sharing/Indexing. Terms are perfectly shared within LEOPARD, meaning that each term is only constructed once and then reused between different occurrences. This does not only reduce 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 symbols, subterm occurrences, etc.) from the term indexing mechanism, terms are kept in β-normal η-long form.

Suite of Normalization Strategies. LEOPARD 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. η-normalization is invariant wrt. β-normalization of spine terms and hence η-normalization (to long form) is applied only once for each freshly created term.

Evaluation and Findings. A recent empirical evaluation [17] has shown that there is *no single best reduction strategy* for HO-ATPs. More precisely, for different TPTP problem categories this study identified different best reduction strategies. This motivates future work in which machine learning techniques could be used to suggest suitable strategies.

3 Multi-agent Blackboard Architecture

In addition to supporting classical, sequential theorem proving procedures, LEOPARD offers means for breaking the global ATP loop down into a set of subtasks that can be computed in parallel. This also includes support for subprover parallelism as successfully employed, for example, in Isabelle/HOL's Sledgehammer tool [7]. More generally, LEOPARD is construed to enable parallalism at various levels inside an ATP, including the term, clause and search level [8]. For this, LEOPARD provides a flexible multi-agent blackboard architecture.

Blackboard Architecture. Process communication in LEOPARD is realized indirectly via a blackboard architecture [20]. The LEOPARD blackboard [21] is a collection of globally shared and accessible data structures which any process, i.e. agent, can query and manipulate at any time in parallel. From the blackboard's perspective each process is a specialist responsible for exactly one kind of problem.

The LEOPARD blackboard mechanism and associated data structures provide specific support for nested and-or search trees, meaning that sets of formulae can be split into (nested) and-or contexts. Moreover, for each supercontext respective TPTP SZS status [18] information is automatically inferred from the statuses of its subcontexts.

Agents. In LEOPARD specialist processes can be modeled as agents [21]. Classically, agents are composed of three components: environment perception, decision making and action execution [20].

The perception of LeoPARD agents is trigger-based, meaning that each agent is notified by a change in the blackboard. LeoPARD agents are to be seen as homomorphisms on the blackboard data together with a filter when to apply an action. They decide upon the perceived change resp. state of the blackboard on the concrete action they want to execute.

Auction Scheduler. Action execution in LeoPARD is coordinated by an auction based scheduler, which implements an own approximation algorithm [21] for combinatorical auctions [2]. More precisely, each LeoPARD agent computes and places a bid for the execution of its action(s). The auction based scheduler then tries to maximize the global benefit of the particular set of actions to choose.

This selection mechanism works uniformly for all agents that can be implemented in LeoPARD. Balancing the value of the actions is therefore crucial for the performance and the termination of the overall system. A possible generic solution for the agents bidding is to apply machine learning techniques to optimize the bids for the best overall performance. This is future work.

Note that the use of advanced agent technology in LeoPARD is optional. A traditional ATP can still be implemented, for example, as a single, sequential reasoner instantiating exactly one agent in the LeoPARD framework.

Agent Implementation Examples. For illustration purposes, some agent implementations have been exemplarily included in the LeoPARD package. For example, simple agents for *simplification*, *skolemization*, *prenex-form*, *negation-normal-form* and *paramodulation* are provided. Moreover, the agent-based integration of external ATPs is demonstrated and their parallelization is enabled by the LeoPARD agent framework. This includes agents embodying LEO-II and Satallax [9] running remotely at the SystemOnTPTP [18] servers in Miami. These example agents can be easily adapted for other TPTP compliant ATPs.

Each example agent comes with an applicability filter, an action definition and an auction value computation. The provided agents suffice to illustrate the working principles of the LeoPARD multi-agent blackboard architecture to interested implementors. After the official release of Leo-III, further, more sophisticated agents will be included and offered for academic reuse.

4 Other Components

The LeoPARD framework provides useful further components. For example, a generic parser is provided that supports all TPTP syntax dialects. Moreover, a command line interpreter supports fine grained interaction with the system. This is useful not only for debugging but also for training and demonstration purposes. As pointed at above, useful support is also provided for the integration of external reasoners based on the TPTP infrastructure. This also includes comprehensive support for the TPTP SZS result ontology. Moreover, ongoing and future work aims at generic means for the transformation and integration of (external) proof protocols, ideally by exploiting results of projects such as ProofCert[2].

[2] See https://team.inria.fr/parsifal/proofcert/

References

1. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. In *Proc. of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 31–46, New York, NY, USA, 1990. ACM.
2. K.J. Arrow. *Social Choice and Individual Values*. Wiley, New York, 1951.
3. C. Benzmüller and M. Kohlhase. LEO – a higher-order theorem prover. In *Proc. of CADE-15*, number 1421 in LNCS, pages 139–143. Springer, 1998.
4. C. Benzmüller and V. Sorge. OANTS – combining interactive and automated theorem proving. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning*, pages 81–97. A.K.Peters, 2001.
5. C. Benzmüller, V. Sorge, M. Jamnik, and M. Kerber. Combined reasoning by automated cooperation. *J. of Applied Logic*, 6(3):318–342, 2008.
6. C. Benzmüller, F. Theiss, L. Paulson, and A. Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic (system description). In *Proc. of IJCAR 2008*, volume 5195 of *LNCS*, pages 162–170. Springer, 2008.
7. J. Blanchette, S. Böhme, and L. Paulson. Extending Sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.
8. M.P. Bonacina. A taxonomy of parallel strategies for deduction. *Annals of Mathematics and Artificial Intelligence*, 29(1–4):223–257, 2000.
9. C.E. Brown. Satallax: An automatic higher-order prover. In *Automated Reasoning*, volume 7364 of *LNCS*, pages 111–117. Springer Berlin Heidelberg, 2012.
10. 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.
11. I. Cervesato and F. Pfenning. A linear spine calculus. *J. Logic and Computation*, 13(5):639–688, 2003.
12. J.Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Paris VII, 1972.
13. A. J. Kfoury, S. Ronchi Della Rocca, J. Tiuryn, and P. Urzyczyn. Alpha-conversion and typability. *Inf. Comput.*, 150(1):1–21, 1999.
14. J. C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, volume 19 of *LNCS*, pages 408–423. Springer, 1974.
15. A. Riazanov. *Implementing an efficient theorem prover*. PhD thesis, University of Manchester, 2003.
16. R. Sekar, I. V. Ramakrishnan, and A. Voronkov. Term indexing. In *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 2001.
17. A. Steen. Efficient data structures for automated theorem proving in expressive higher-order logics. Master's thesis, Freie Universität Berlin, 2014. http://userpage.fu-berlin.de/∼lex/drop/steen_datastructures.pdf.
18. G. Sutcliffe. The TPTP problem library and associated infrastructure. *J. Automated Reasoning*, 43(4):337–362, 2009.
19. G. Sutcliffe and C. Benzmüller. Automated reasoning in higher-order logic using the TPTP THF infrastructure. *J. Formalized Reasoning*, 3(1):1–27, 2010.
20. Gerhard Weiss, editor. *Multiagent Systems*. MIT Press, 2013.
21. M. Wisniewski. Agent-based blackboard architecture for a higher-order theorem prover. Master's thesis, Freie Universität Berlin, 2014. http://userpage.fu-berlin.de/∼lex/drop/wisniewski_architecture.pdf.