Strategies and Machine Learning for Lash

Chad E. Brown[1], Jan Jakubuv[1,2], and Cezary Kaliszyk[2]

[1] Czech Technical University in Prague, Czechia
[2] University of Innsbruck, Austria

Lash [BK22] is an automated theorem prover for higher-order logic. Lash is a fork of the theorem prover Satallax [Bro12; FB16]. Lash replaces Satallax's Ocaml representation of terms with an efficient representation of normal terms (with unique integer ids) in C. The representation in C also stores useful information such as which de Bruijn indices are free in the term. Knowing the free de Bruijn indices of terms makes recognizing potential η-redexes possible without traversing the λ-abstraction. Likewise it is possible to determine when shifting and substitution of de Bruijn indices would not affect a term, avoiding the need to traverse the term. Computations such as substitutions and shifting de Bruijn indices are cached to prevent recomputation.

In addition to the low-level C term reimplementation, we have also provided a number of other low-level functionalities replacing the slower parts of the Ocaml code. This includes low-level priority queues, as well as C code used to associate the integers representing normal propositions with integers that are used to communicate with MiniSat.

As with Satallax, the nature of Lash's search is highly dependent on the settings of boolean and integer flags. A *mode* (also called a *strategy*) is a collection of flag settings. A *schedule* is a sequence of modes, along with a timeout. A few of the important flags include:

- Initial_Subterms_As_Instantiations is a boolean flag. If set to true, Lash uses closed subterms of the initial problem as instantiations.
- Eagerly_Process_Instantiations is a boolean flag. If set to true, new instantiations are processed immediately instead of being put onto the priority queue.
- Split_Global_Disjunctions is a boolean flag. If set to true, Lash tries to split the problem into several subproblems to be proven independently before beginning the search.
- Minisat_Search_Period is an integer flag that controls how often Minisat is asked to search for a model of the current set of propositional clauses. Sometimes this is set to a low number, e.g., 10, so that Minisat checks for unsatisfiability after every tenth step. Often it is useful to set the flag to a very high number, e.g., a billion, so that Lash will effectively not ask Minisat to check for unsatisfiability, unless all other options are exhausted. (In many problems Minisat determines unsatisfiability via unit propagation without decisions.)
- Enum_Start is an integer flag that determines how long to delay beginning enumeration of instantiations for quantifiers at function types. If a problem has a higher-order quantifier that is irrelevant to the problem, then a high value is helpful (since no higher-order instantiation is needed). If the problem requires instantiating a higher-order quantifier, then a lower value is more likely to lead to success.
- Axiom_Delay is an integer flag that (if nonzero) nudges Lash to work on the negated conjecture before working on the axioms, with larger integers corresponding to longer delays.

Following Satallax, Lash reads a problem and determines if it exceeds a certain size threshold, classifying problems as "small" or "large." For large problems an implementation of SInE [HV11] reduces the size of the problem and then searches using a schedule appropriate for large problems. For small problems a search proceeds using a schedule appropriate for small problems. Which modes and schedules are appropriate for which kinds of problems is determined experimentally.

Starting with several manually designed strategies (i.e., modes) and we use the strategy invention system Grackle[1] to invent more strategies targeted to randomly selected 1800 TPTP THF problems. Grackle is a generalization of strategy invention system BliStr [Urb13]. Both systems are based an evolutionary algorithm, where strategies are considered "animals" and problems to be solved their "food". Only animals that consume enough food, that is, solve enough problems, survive to the next generation and are given a chance to conceive an offspring. This algorithm favors animals that consume food not consumed by others. This leads to diversity and complementarity of invented strategies.

While BliStr is a strategy invention system for E Prover, Grackle can be instantiated to invent strategies for any solver. A requisite of this instantiation is a parametrization of the solver configuration space, in our case, the collection of Lash mode flags. We start by extracting flag names and possible option values from the collection of manually designed modes. This gives us 130 flag names. Out of them, 43 correspond to boolean options. The rest are integer options, with domains ranging in size from 2 to 26. Altogether, the space contains around 10^{60} possible configurations.

Once the configurations space is parametrized, we can launch Grackle with 10 initial configurations, selected from the manually designed strategies as the most complementary and strongest ones. During the strategy invention, configurations are evaluated with a short runtime limit of 1 second. The best of the initial strategies solves 285 problems (out of 1800 TPTP problems) in 1 second. Together, the 10 initial strategies solve 358 problems.

We limit Grackle runs to 24 hours to be able to evaluate several strategy invention options. We try different settings for maximal strategy generation size, and several variations of strategy specialization used to produce offspring. Together we run around 14 Grackle runs, each running on 8 CPU cores. All the runs produce more than 3000 different modes, the best of them solves 355 problems in 1 second. The greedy collection of 10 best new modes solves together 449 problems, and the total coverage of all the modes is 489.

We construct a Lash *schedule* by evaluating best 110 strategies on the training problems in 30 seconds. From this evaluation, we construct a greedy cover of 10 best strategies. This greedy cover gives us an order in which to run the modes, provided the overall time limit is evenly distributed among the modes. Additionally, we construct a schedule with 20 modes, extending the first 10 with another greedy cover constructed without the previously selected modes. Furthermore, we can also split the results of the evaluation into results on "small" and "large" problems, and we can construct two different schedules for them.

We are investigating the possible ways how machine learning can be included in Lash. In particular, this could involve the extension of the shared C representation by precomputing various features useful for prioritizing the available actions [FB16]. Other ways to apply machine learning include the selection or even generation of interesting instances for a given problem, as well as machine-learning guided interaction with the SAT-solver.

Acknowledgements The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902 and the ERC starting grant no. 714034 SMART.

[1] https://github.com/ai4reason/grackle

References

[BK22] C. E. Brown and C. Kaliszyk. "Lash 1.0 (System Description)". In: accepted for publication at IJCAR 2022. 2022.

[Bro12] C. E. Brown. "Satallax: An Automatic Higher-Order Prover". In: *IJCAR*. Ed. by B. Gramlich, D. Miller, and U. Sattler. Vol. 7364. LNCS. Springer, 2012, pp. 111–117.

[FB16] M. Färber and C. E. Brown. "Internal Guidance for Satallax". In: *Automated Reasoning - 8th International Joint Conference, IJCAR 2016*. Ed. by N. Olivetti and A. Tiwari. Vol. 9706. LNCS. Springer, 2016, pp. 349–361. URL: https://doi.org/10.1007/978-3-319-40229-1_24.

[HV11] K. Hoder and A. Voronkov. "Sine Qua Non for Large Theory Reasoning". In: *23rd International Conference Automated Deduction - CADE-23*. Ed. by N. S. Bjørner and V. Sofronie-Stokkermans. Vol. 6803. LNCS. Springer, 2011, pp. 299–314. DOI: 10.1007/978-3-642-22438-6_23. URL: https://doi.org/10.1007/978-3-642-22438-6_23.

[Urb13] J. Urban. "BliStr: The Blind Strategymaker". In: *CoRR* abs/1301.2683 (2013). arXiv: 1301.2683. URL: http://arxiv.org/abs/1301.2683.