15th International Workshop on the Implementation of Logics

– Preliminary Proceedings –

May 26, 2024, Balaclava, Mauritius

Konstantin Korovin,
Michael Rawson,
Stephan Schulz (eds.)

Foreword

The *15th International Workshop on the Implementation of Logics* was held with the *25th International Conference on Logic for Programming, Artificial Intelligence and Reasoning* (LPAR-25) in Balaclava, on Mauritius in the Indian Ocean. IWIL ran on May 26th, with a program that included one invited talk, one contributed talk, and three paper presentations, followed by a lively panel discussion on the architecture of modern automated reasoning systems.

This volume collects the presented papers.

Konstantin Korovin, Michael Rawson, Stephan Schulz

IWIL-24 Program

- Tanel Tammet (Tallinn University of Technology, Estonia): Invited talk: *Hash Indexes for Resolution-based FOL Provers*
- Daniel Ranalter, Cezary Kaliszyk (University of Innsbruck, Austria): *User-aided Conjecturing for Automated Theorem Proving*
- Stephan Schulz (DHBW Stuttgart, Germany): *Shared Terms and Cached Rewriting*
- Jack McKeown, Geoff Sutcliffe (University of Miami, United States): *Dataset-Specific Strategies for the E Theorem Prover*
- David Fuenmayor (University of Bamberg, Germany), Jack McKeown, Geoff Sutcliffe (University of Miami, United States): *Towards StarExec in the Cloud*
- IWIL Panel: *Software Architectures for Modern Automated Reasoning Systems - Lessons Learned?*
  - Nikolaj Bjørner (Microsoft Research)
  - Katalin Fazekas (Technical University of Vienna)
  - Michael Rawson (Technical University of Vienna)
  - Stephan Schulz (DHBW Stuttgart)
  - Tanel Tammet (Tallinn University of Technology, Estonia)
  - Moderation: Konstantin Korovin (University of Manchester)

Dataset-Specific Strategies for the E Theorem Prover

Jack McKeown and Geoff Sutcliffe

University of Miami, Miami, Florida, U.S.A.
jam771@miami.edu, geoff@cs.miami.edu

Abstract

The E automated theorem proving system has an "automatic" mode that analyzes the input problem in order to choose an effective proof search strategy. A strategy includes the term/literal orderings, given clause selection heuristics, and a number of other parameters. This paper investigates the idea of creating one strategy for a given dataset of problems by merging the strategies chosen by E's automatic mode over all of the problems in the dataset. This strategy merging approach is evaluated on the MPTPTP2078, VBT, and SLH-29 datasets. Surprisingly, the merged strategies outperform E's automatic mode over all three datasets.

1 Introduction

The core component of many saturation-based Automated Theorem Proving (ATP) systems is the "given clause" algorithm [11]. This algorithm maintains two sets of clauses: a *processed* set that is initially empty, and an *unprocessed* set that initially contains the clauses from the axioms and negated conjecture. One at a time, a *given clause* is selected from the unprocessed set and brought into the processed set, then inferences are made between the given clause and other clauses in the processed set. The inferred clauses are added to the unprocessed set modulo redundancy criteria [8]. This process repeats until the empty clause is derived, the unprocessed set becomes empty, or a resource limit is reached. The derivation of the empty clause indicates that the conjecture is a theorem of the axioms, whereas an empty unprocessed set indicates that the conjecture is not a theorem of the axioms.

The saturation-based ATP system E [9] implements the DISCOUNT version [3] of the given clause algorithm. E has an automatic mode that analyzes the input problem in order to choose an effective proof search strategy. Currently, an E strategy consists of 108 key-value pairs, for the various parameters that influence the proof search. This paper investigates the idea of merging the strategies chosen by E's automatic mode for a set of problems into a single merged strategy, and using that merged strategy for all the problems.

Section 2 briefly summarizes how E performs given clause selection, how its given clause selection can be controlled by users, and how this control is defined in an E strategy. Section 3 describes how the strategy merging is performed, and describes how the merged strategies are evaluated. Section 4 describes the datasets used for evaluation, gives details about the experiments performed, and presents the experimental results. Section 5 summarizes the results, and concludes the paper.

2 Given Clause Selection in E

In E, given clause selection is guided by *clause evaluation functions (CEFs)*. Each CEF evaluates each unprocessed clause, determining a priority for each clause in a priority queue associated with the CEF. E supports a number of different CEFs, each of which is composed of an instance of a *weight function* that evaluates the clause, and a *priority function* that restricts the

scope of the CEF, (Each priority function partitions the clauses in the unprocessed set so that a certain class of clauses is given preference regardless of the evaluations given by the weight function). Each weight function has a set of parameters unique to that weight function, which are provided after the priority function. Figure 1 shows an example CEF with its components labeled.

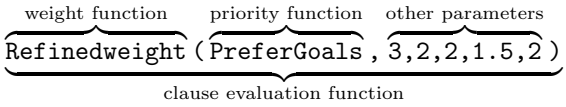

Figure 1: Example of an E clause evaluation function

During proof search CEFs are used to select the given clauses according to a *heuristic*. A heuristic is an ordered list of CEFs, each having its own integer *heuristic weight* that determines how many clauses should be selected from that CEF's priority queue before moving on to the next CEF (or back to the first CEF after the last CEF in the heuristic). Figure 2 shows an example of an E heuristic, with each line showing a CEF prefixed by its heuristic weight: the first CEF would be used once to select the given clause, then the second CEF would be used four times, then the third CEF would be used ten times, etc. This schedule would then repeat after $1 + 4 + 10 + 3 + 5 = 23$ given clause selections. The heuristic is only one part of a full E strategy. A full strategy with all 108 strategy parameters is shown in Appendix A. E strategies include a heuristic like the one shown in Figure 2 as the value of the `heuristic_def` key.

```
(1.ConjectureRelativeSymbolWeight(SimulateSOS,0.5,100,100,100,100,1.5,1.5,1),
4.ConjectureRelativeSymbolWeight(ConstPrio,0.1,100,100,100,100,1.5,1.5,1.5),
10.FIFOWeight(PreferProcessed),
3.ConjectureRelativeSymbolWeight(PreferNonGoals,0.5,100,100,100,100,1.5,1.5,1),
5.Refinedweight(SimulateSOS,3,2,2,1.5,2))
```

Figure 2: Example of an E heuristic.

E's automatic mode for choosing a strategy is invoked using the `--auto` flag, and if invoked with the `--print-strategy` flag, E will print out the strategy in the format shown in Appendix A. Therefore, an E strategy can be saved to a file by invoking E with the `--auto` and `--print-strategy` flags and redirecting `stdout` to a file. The format of these files is similar to JSON.

3 Strategy Merging

In this work the strategies chosen by `--auto` for every problem in a given dataset are saved without attempting to solve the problems, and these saved strategies are merged in multiple ways to create other strategies that are used to solve all of the problems. This primarily means creating merged strategies that are each evaluated on all problems, but it also means creating per-problem strategies via merging. For all 107 strategy parameters other than the `heuristic_def`, the value used in the merged strategy is the value that was used most frequently in the individual strategies. The `heuristic_def` is merged in a more sophisticated way, as follows.

This paper evaluates three ways of merging the `heuristic_def` parameter:

1. The simplest way to merge the heuristics takes the union of the sets of CEFs used in all the saved strategies, and assigns a heuristic weight of 1 to each of them. The E strategy that results from merging heuristics in this way is called *MasterAllOnes*. This approach ignores the heuristic weights in the saved strategies as well as the number of saved strategies that each CEF occurs in.

2. The second way to merge the heuristics is to assign to each CEF a heuristic weight proportional to the sum of its heuristic weights from all of the saved strategies. The sums are not used directly because having very large heuristic weights would cause E to repeatedly ignore important clauses that are not preferred by a CEF being repeatedly used, but are preferred by other CEFs. Therefore the sums are scaled down by a constant factor and then rounded up using the `ceil` function. The scaling factor is determined so that the maximum heuristic weight is 20. This number was chosen as a middle ground between no scaling and aggressive scaling that would lose more information about the distribution of CEFs due to the rounding. The `ceil` function guarantees that no CEF is removed entirely from the merged heuristic. The E strategy that results from merging heuristics this way is called *MasterWeighted*.

3. To evaluate the impact of MasterWeighted's repeated use of the same CEFs, the MasterWeighted strategy is also evaluated using a modified version of E that attempts to avoid consecutively using the same CEF. It does this by going through all CEFs in a round-robin fashion. Each CEF gets a counter is initialized at its heuristic weight, and this counter is decremented when that CEF is used. When the counter reaches zero, that CEF is skipped until the heuristic resets. Once all counters reach zero, they are all reset to their heuristic weights. This is easiest to understand by example. Originally, the heuristic "3*CEF1, 2*CEF2, 1*CEF3" leads to the the following sequence of CEFs used for selection (before repeating): "1,1,1,2,2,3." Under the modified version of E, the same heuristic leads to the following sequence instead: "1,2,3,1,2,1." This method is called *MasterWeightedRR*.

4. Lastly, a version of the MasterWeighted strategy is created by using only the saved strategies for problems that `--auto` was able to solve. This strategy, referred to as *MasterSuccess*, was created with the intuition that the strategies suggested by `--auto` should only be trusted to contribute to the merged strategy if they were successful on the problem for which they were suggested.

In all of the strategies, the CEFs in the merged heuristic appear in order of decreasing heuristic weight so that the "best" CEFs are first. In MasterAllOnes, where all heuristic weights are set to 1, the order is the same as in MasterWeighted.

3.1 Potential ITP Application

While the strategy merging described above could be helpful when dealing with a fixed dataset of problems, it would also be useful if a merged strategy could be evolved and applied incrementally for a growing set of related problems. This situation is encountered when ATP systems are used as "hammers" in Interactive Theorem Proving (ITP) systems [5]. A well-known example is the Isabelle [7] ITP system, whose "Sledgehammer" mode [6] submits subproblems to ATP systems like E.

5. To evaluate the potential of strategy merging for a growing set of problems, another set of strategies was created, collectively referred to as *MasterIncremental*. These strategies are

created in the order that the problems are added to the set, with the kth problem getting assigned a merged strategy formed from the `--auto` strategies of the first k problems. The strategy assigned to the first problem is the same as its `--auto` strategy, and the strategy assigned to the last problem is the same as MasterWeighted. Each merging is done the same as in MasterWeighted, only with different sets of input strategies.

3.2 Ablation Study

The `heuristic_def` parameter was hypothesized to have a larger impact on the results than the other parameters in merged strategies because given clause selection is the core of the proof search. To test this hypothesis, two other methods for strategy merging were evaluated.

6. The first method, called *CommonHeuristic*, sets the `heuristic_def` parameter to its value in the MasterWeighted strategy, but keeps the value chosen by `--auto` for all the other 107 strategy parameters.

7. The second method, called *CommonElse*, is essentially the converse, keeping the value chosen by `--auto` for the `heuristic_def` parameter but setting the other 107 strategy parameters to their values in the MasterWeighted strategy.

3.3 An Auto-based Baseline

While the strategy merging can produce a strategy that generally outperforms E's automatic strategies, it is unclear whether this is due to the merged strategy being better than all of the `--auto` strategies, or if E's `--auto` mode is assigning suboptimal strategies from its set of available strategies.

8. To test this, every unique strategy that E's `--auto` mode assigns over all problems in a dataset is evaluated on all problems in the dataset. A baseline method called *AutoAll* is created by picking the best performing strategy for each problem in the dataset. Therefore, if even one strategy solves a problem, then AutoAll solves that problem. This simulates how good E's `--auto` mode could be if it perfectly picked the best strategy for each problem (from its set of available strategies).

4 Data, Experiments, and Results

All in all, nine methods for solving a set of problems were evaluated: `--auto`, AutoAll, MasterAllOnes, MasterWeighted, MasterWeightedRR, MasterSuccess, MasterIncremental, CommonHeuristic, and CommonElse. MasterAllOnes, MasterWeighted, MasterWeightedRR, and MasterSuccess each consist of a single E strategy, whereas the other methods have a different strategy for each problem.

All methods were evaluated on three datasets: MPTPTP2078, VBT, and SLH-29. The MPTPTP2078 dataset is a TPTP-compliant version of the MPTP2078 dataset [1] that consists of problems formed from the derivation of the Bolzano-Weierstrass theorem in the Mizar Mathematical Library [4]. These problems come in "bushy" and "chainy" variants, with the bushy variants having only the most immediately relevant axioms and the chainy variants having a larger set of axioms. The "bushy" variants of the problems were used in this work. The VBT and SLH-29 datasets were both used in the CASC-J11 competition [10], and come from the

Sledgehammer mode of the Isabelle theorem prover. The VBT dataset consists of 8000 problems generated by Isabelle's Sledgehammer mode from the Van Emde Boas Trees entry in the Isabelle Archive of Formal Proofs [2]. The problems are available in multiple logics, and the typed-first order versions were used here. The SLH-29 dataset is a collection of 8400 higher-order problems that also come from interactions with the Sledgehammer mode in Isabelle. For most strategy parameters the `--auto` setting is largely consistent across problems within each dataset. For example, in the MPTPTP2078, VBT, and SLH-29 datasets, only 16, 3, and 27 strategy parameters, respectively, had two or more values used in at least 5% of the problems.

The strategy merging and experimental setup are diagramed in Figure 3. The process is the same for all datasets:

1. E is used to save strategies for each problem using `--auto --print-strategy`.
2. The saved strategies are merged in the ways described above to get the strategies for MasterAllOnes, MasterWeighted, and MasterIncremental.
3. CommonHeuristic and CommonElse strategies are created for each problem by taking some parameter values from the saved per-problem strategies and others from the MasterWeighted strategy.
4. E is invoked on all problems using the `--auto` flag. The set of solved problems is used to select the saved strategies from step 1 that are then used to make the MasterSuccess strategy.
5. E is invoked on all problems using the MasterAllOnes, MasterWeighted, MasterSuccess, MasterIncremental, CommonHeuristic, and CommonElse methods.
6. Every strategy suggested by `--auto` is used for every problem to get the AutoAll results.

Strategies are loaded into E using the `--parse-strategy` flag, and every call to E includes the flags `--soft-cpu-limit=60` and `--cpu-limit=65`, which limit CPU time.

The results are shown in Tables 1 and 2. Table 1 shows the number of problems solved by each method. Table 2 shows the median number of given clauses selected before finding a proof, for only problems solved by all methods and excluding problems solved during presaturation interreduction (which is not guided by a strategy). In both tables the best result for each dataset is bolded. Because AutoAll is the clear winner in terms of solved problems and processed clauses for all datasets, the second best results are also bolded.

Dataset	`--auto`	Auto All	Master AllOnes	Master Weighted	Master Weighted RR	Master Success	Master Incr.	Common Heuristic	Common Else
M'2078	1151	**1438**	**1219**	1210	1201	1199	1199	1170	1086
VBT	2637	**3596**	2701	2841	**2858**	2806	2785	2710	2521
SLH-29	3396	**4203**	3642	**3743**	3565	3505	3556	3430	3371

Table 1: Problems solved by each method

An additional perspective on the results is given by Figures 4, 5, and 6. The vertical lines in these figures show the same information as Table 1. Each row of the black and white background represents one strategy, and each column represents one problem. The rows and

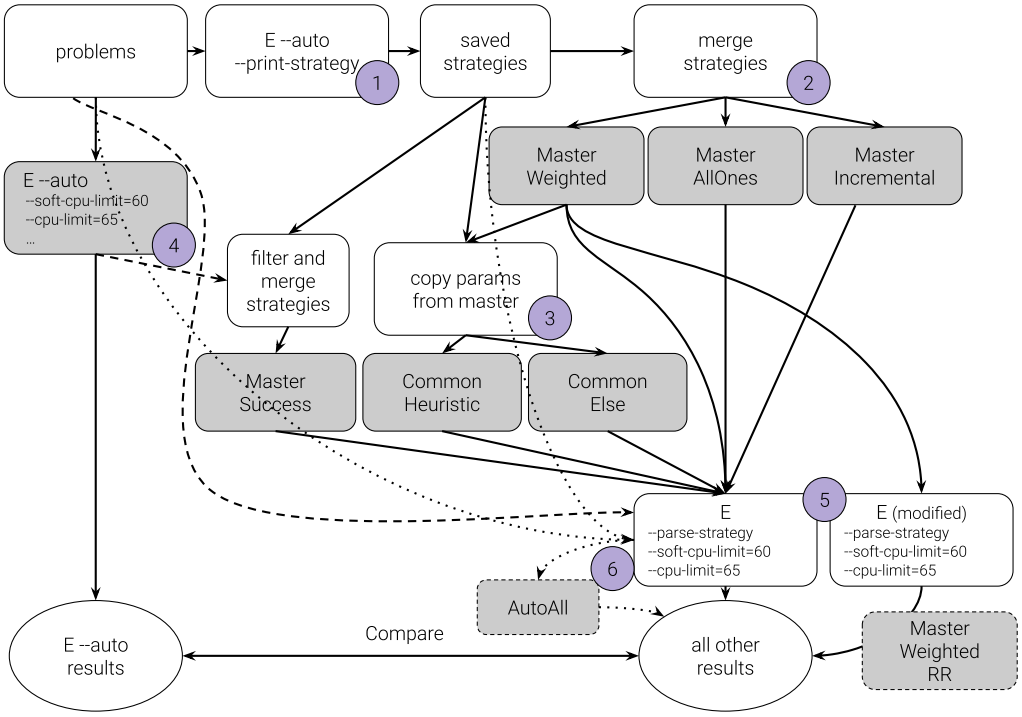

Figure 3: Strategy merging and experimental setup

Dataset	`--auto`	Auto All	Master AllOnes	Master Weighted	Master Weighted RR	Master Success	Master Incr.	Common Heuristic	Common Else
M'2078	104	**51.5**	**92**	103.5	119.5	117	112	97	105
VBT	1794.5	**693.5**	**1475**	1565	1524	1600.5	1558	1531	1878
SLH-29	1431	**492**	874.5	**751**	827.5	760	757	932	1038

Table 2: Median number of clauses processed before finding a proof.
(For only problems solved by all methods)

columns are sorted such that the strategy that solves the most problems appears on top and the problems that are solved by the most strategies appear on the left. For instance, the fact that the transition from black to white occurs earlier in Figure 5 than in Figure 4 suggests that the problems in the VBT dataset are harder on average than the problems in the MPTPTP2078 dataset.

The MasterAllOnes and MasterWeighted strategies both improve upon `--auto` in terms of number of problems solved and processed clauses on all datasets. The MasterWeighted strategy solves more problems than the MasterAllOnes strategy on the VBT and SLH-29 datasets, but not on the MPTPTP2078 dataset. Perhaps this is because the MPTPTP2078 problems are solved in fewer given clause selections on average. The MasterAllOnes strategy uses more unique CEFs in the short-term, but the MasterWeighted strategy ostensibly uses the CEFs in

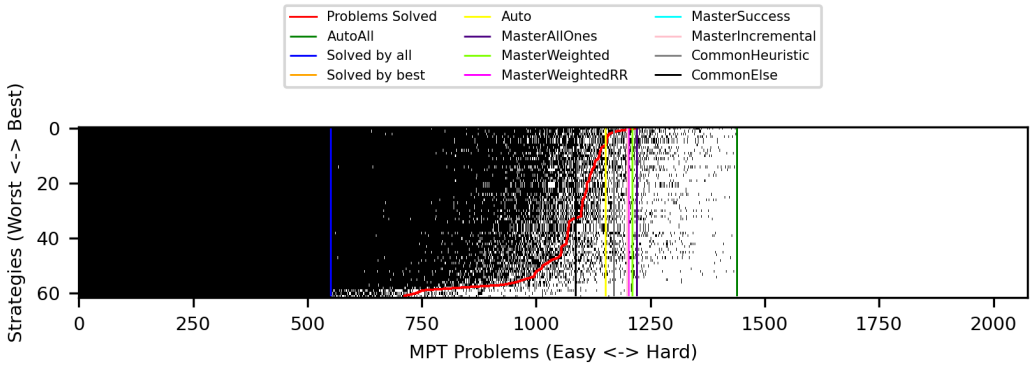

Figure 4: MPTPTP2078 Experiment Results

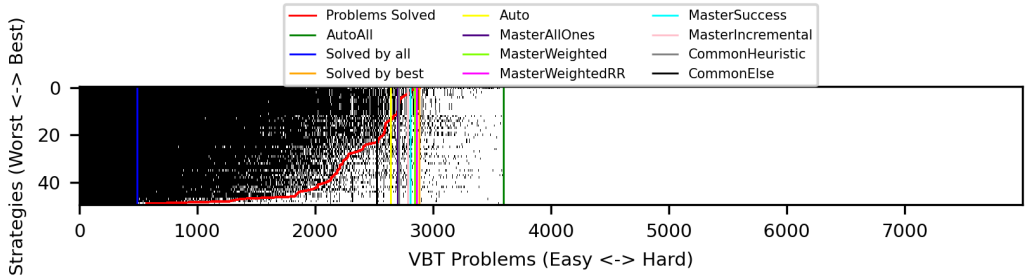

Figure 5: VBT Experiment Results

better proportions in the long-term. Even if many CEFs agree that an important clause should be selected, the repeated use of a single CEF in the MasterWeighted strategy could delay the clause's selection. In such a case, the MasterAllOnes strategy would select the important clause more quickly. As problem difficulty increases, however, this potential delay would represent a smaller proportion of the total selections needed to find a proof. This was the motivation behind the MasterWeightedRR method. The results were mixed, however, with MasterWeightedRR solving fewer problems than both MasterWeighted and MasterAllOnes on the MPTPTP2078 and SLH datasets, but more than both the VBT dataset.

The AutoAll results suggest that E's auto mode could be improved by selecting a more effective strategy for each problem without modifying the underlying set of candidate strategies that E's `--auto` mode uses. Additionally, the AutoAll result provides context that merged strategies are not universally better than the individual strategies, although they are better on average than the particular ones chosen by E's `--auto` mode.

The MasterSuccess strategies perform worse than both the MasterWeighted strategy and MasterAllOnes strategy in terms of both number of problems solved and processed clauses. This is surprising because this strategy is constructed by merging only the strategies that were successful in solving their associated problem. Perhaps failure to solve a problem is more of an indication that the problem is difficult than it is an indication that the strategy chosen by `--auto` is bad.

The MasterIncremental strategies perform worse than MasterWeighted and MasterAllOnes

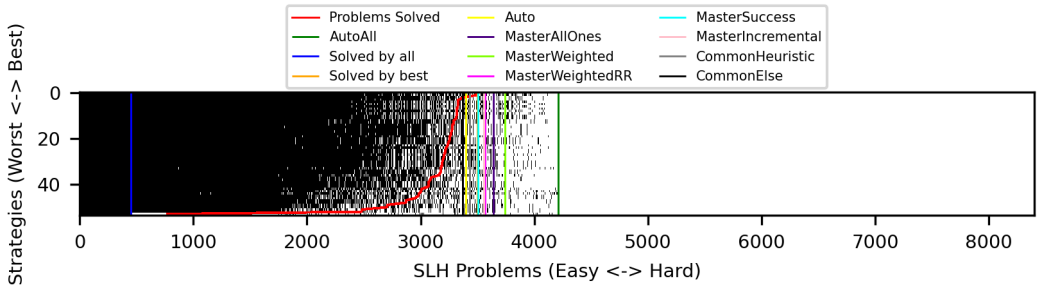

Figure 6: SLH Experiment Results

on all datasets in terms of both number of problems solved and processed clauses. In light of the general success of strategy merging, this was unsurprising because fewer strategies are being merged to create each MasterIncremental strategy than were merged in MasterAllOnes or MasterWeighted. That being said, MasterIncremental solves more problems than `--auto` on all three datasets and uses fewer processed clauses (median) on the VBT and SLH-29 datasets, suggesting that incremental strategy merging could be useful within ITP "hammers".

The CommonHeuristic strategies outperform the `--auto` strategies on each dataset, whereas the CommonElse strategies do not, except for in terms of the number of clauses processed on the SLH-29 dataset. This suggests a coupling between the 107 merged non-`heuristic_def` strategy parameters and the merged `heuristic_def` parameter. The merged non-`heuristic_def` parameter values are beneficial, but only when used in conjunction with the merged `heuristic_def` parameter. (It cannot be the case that the merged heuristic is the only helpful merged parameter, because MasterWeighted outperforms CommonHeuristic.)

5 Conclusion

This paper demonstrates that, at least for the three datasets used here, it is possible to improve upon E's automatic strategy by merging the strategies that E automatically chooses, and then using the merged strategy for all of the problems. While this approach would likely be less effective over a very diverse dataset, this strategy merging seems to be a helpful way to inject helpful bias for a homogenous dataset. Additionally, incremental strategy merging shows promise for incorporation into ITP tools like Sledgehammer.

References

[1] J. Alama, D. Kühlwein, E. Tsivtsivadze, J. Urban, and T. Heskes. Premise Selection for Mathematics by Corpus Analysis and Kernel Methods. *CoRR*, abs/1108.3446, 2011.

[2] T. Ammer and P. Lammich. van Emde Boas Trees. *Archive of Formal Proofs*, November 2021. https://isa-afp.org/entries/Van_Emde_Boas_Trees.html, Formal proof development.

[3] Jürgen Avenhaus, Jörg Denzinger, and Matthias Fuchs. DISCOUNT: A System for Distributed Equational Deduction. In *Rewriting Techniques and Applications: 6th International Conference, RTA-95 Kaiserslautern, Germany, April 5–7, 1995 Proceedings 6*, pages 397–402. Springer, 1995.

[4] G. Bancerek, C. Bylinski, A. Grabowski, A. Kornilowicz, R. Matuszewski, A. Naumowicz, K. Pak, and J. Urban. Mizar: State-of-the-art and Beyond. In *Intelligent Computer Mathematics - In-*

ternational Conference, CICM July 13-17, 2015, Proceedings, volume 9150 of *Lecture Notes in Computer Science*, pages 261–279. Springer, 2015.

[5] J. Blanchette, C. Kaliszyk, L. Paulson, and J. Urban. Hammering Towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

[6] Jia Meng and Lawrence C. Paulson. Translating Higher-Order Clauses to First-Order Clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.

[7] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283. Springer Science & Business Media, 2002.

[8] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[9] S. Schulz, S. Cruanes, and P. Vukmirovic. Faster, Higher, Stronger: E 2.3. In *Proceedings of the 27th International Conference on Automated Deduction*, number 11716 in Lecture Notes in Computer Science, pages 495–507. Springer-Verlag, 2019.

[10] G. Sutcliffe and M. Desharnais. The 11th IJCAR Automated Theorem Proving System Competition - CASC-J11. *AI Commun.*, 36(2):73–91, 2023.

[11] A' Voronkov. Algorithms, Datastructures, and Other Issues in Efficient Automated Deduction. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 2083 in Lecture Notes in Artificial Intelligence, pages 13–28. Springer-Verlag, 2001.

A Example Strategy

Here is the strategy chosen by E for the MPT0001+1.p problem from the MPTPTP2078 dataset, given as an example of a strategy file. The whitespace around the heuristic has been adjusted for readability, so it might not work within E without edits.

```
{
  {
     ordertype:               KBO6
     to_weight_gen:           precedence
     to_prec_gen:             invfreqhack
     rewrite_strong_rhs_inst: true
     to_pre_prec:             ""
     conj_only_mod:           0
     conj_axiom_mod:          0
     axiom_only_mod:          0
     skolem_mod:              0
     defpred_mod:             0
     force_kbo_var_weight:    false
     to_pre_weights:          ""
     to_const_weight:         0
     to_defs_min:             false
     lit_cmp:                 1
     lam_w:                   20
     db_w:                    10
     ho_order_kind:           lfho
  }
  no_preproc:                     false
  eqdef_maxclauses:               20000
  eqdef_incrlimit:                20
  formula_def_limit:              24
```

```
sine:                           "Auto"
add_goal_defs_pos:             false
add_goal_defs_neg:             false
add_goal_defs_subterms:        false
heuristic_name: Default
heuristic_def: "(
  1.ConjectureRelativeSymbolWeight(SimulateSOS,0.5,100,100,100,100,1.5,1.5,1),
  4.ConjectureRelativeSymbolWeight(ConstPrio,0.1,100,100,100,100,1.5,1.5,1.5),
  1.FIFOWeight(PreferProcessed),
  1.ConjectureRelativeSymbolWeight(PreferNonGoals,0.5,100,100,100,100,1.5,1.5,1),
  4.Refinedweight(SimulateSOS,3,2,2,1.5,2)
)"
prefer_initial_clauses:         false
selection_strategy:             SelectComplexExceptUniqMaxHorn
pos_lit_sel_min:                0
pos_lit_sel_max:                9223372036854775807
neg_lit_sel_min:                0
neg_lit_sel_max:                9223372036854775807
all_lit_sel_min:                0
all_lit_sel_max:                9223372036854775807
weight_sel_min:                 0
select_on_proc_only:            false
inherit_paramod_lit:            false
inherit_goal_pm_lit:            false
inherit_conj_pm_lit:            false
enable_eq_factoring:            true
enable_neg_unit_paramod:        true
enable_given_forward_simpl:     true
pm_type:                        ParamodSim
ac_handling:                    1
ac_res_aggressive:              true
forward_context_sr:             true
forward_context_sr_aggressive:  false
backward_context_sr:            false
forward_subsumption_aggressive: false
forward_demod:                  2
prefer_general:                 false
condensing:                     false
condensing_aggressive:          false
er_varlit_destructive:          true
er_strong_destructive:          true
er_aggressive:                  true
split_clauses:                  0
split_method:                   0
split_aggressive:               false
split_fresh_defs:               true
rw_bw_index_type:               FP7
pm_from_index_type:             FP7
pm_into_index_type:             FP7
sat_check_grounding:            ConjMinMinFreq
sat_check_step_limit:           5000
sat_check_size_limit:           9223372036854775807
sat_check_ttinsert_limit:       9223372036854775807
```

```
    sat_check_normconst:            false
    sat_check_normalize:            false
    sat_check_decision_limit:       10000
    filter_orphans_limit:           9223372036854775807
    forward_contract_limit:         9223372036854775807
    delete_bad_limit:               2000000000
    mem_limit:                      0
    watchlist_simplify:             true
    watchlist_is_static:            false
    use_tptp_sos:                   false
    presat_interreduction:          true
    detsort_bw_rw:                  false
    detsort_tmpset:                 false
    arg_cong:                       all
    neg_ext:                        off
    pos_ext:                        off
    ext_rules_max_depth:            -1
    inverse_recognition:            false
    replace_inj_defs:               false
    lift_lambdas:                  true
    lambda_to_forall:              true
    unroll_only_formulas:          true
    elim_leibniz_max_depth:        -1
    prim_enum_mode:                pragmatic
    prim_enum_max_depth:           -1
    inst_choice_max_depth:         -1
    local_rw:                      false
    prune_args:                    false
    preinstantiate_induction:      false
    fool_unroll:                   true
    func_proj_limit:               0
    imit_limit:                    0
    ident_limit:                   0
    elim_limit:                    0
    unif_mode:                     single
    pattern_oracle:                true
    fixpoint_oracle:               true
    max_unifiers:                  4
    max_unif_steps:                256
}
```

Towards StarExec in the Cloud

David Fuenmayor[1], Jack McKeown[2], and Geoff Sutcliffe[2]

[1] University of Bamberg, Bamberg, Germany
david.fuenmayor@uni-bamberg.de
[2] University of Miami, Miami, USA
jam771@miami.edu,geoff@cs.miami.edu

Abstract

StarExec has been central to much progress in logic solvers over the last 10 years. It was recently announced that StarExec Iowa will be decommissioned, and while StarExec Miami will continue to operate while funding is available, it will not be able to support all the logic solver communities currently using the larger StarExec Iowa. In the long term StarExec will necessarily have to migrate to new compute environments. This paper describes work being done to reengineer StarExec as a cloud-native application using container technology and infrastructure-as-code practices. The first step has been to containerise StarExec and ATP systems so that they can be run on a broad range of computer platforms. The next step in process is to write a new backend in StarExec so that Kubernetes can be used to orchestrate distribution of StarExec job pairs over whatever compute nodes are available. Supported by an Amazon Research Award, a new version of StarExec will be deployed in AWS.

1 Introduction

Automated Theorem Proving (ATP) is concerned with the development and use of tools that automate sound reasoning: the derivation of conclusions that follow inevitably from facts. Automated Theorem Proving (ATP) is at the heart of many computational tasks, in particular for verification [12, 10] and security [8].[1] New and emerging application areas include chemistry [44], biology [6], medicine [14], elections [21, 4], auctions [5], privacy [18], law [24], ethics [9], religion [22, 2, 16], and business [11]. ATP systems are also used as components of more complex Artificial Intelligence (AI) systems, and the impact of ATP is thus extended into many facets of society.

The Thousands of Problems for Theorem Provers (TPTP) World [40] is a well established infrastructure that supports research, development, and deployment of ATP systems. The TPTP World includes the TPTP problem library [39], the TSTP solution library [37], standards for writing ATP problems and reporting ATP solutions [41, 36], tools and services for processing ATP problems and solutions [37], and it supports the annual CADE ATP System Competition (CASC) [38]. Since its first release in 1993 the ATP community has used the TPTP World as an appropriate and convenient infrastructure for ATP system development, evaluation, and application. The TPTP World has a diverse, engaged, and sustained user community, with various parts of the TPTP World being deployed in a range of applications in both academia and industry.[2] The web page www.tptp.org provides access to all components.

The TPTP problem library was motivated by the need to provide support for meaningful ATP system evaluation. The need to provide support for meaningful system evaluation has been recognized in many other logic solver communities, e.g., TPTP [42], SAT [15], SMT [7], Termination [19], etc. For many years testing of logic solvers was done on individual developers'

[1] In AWS - aws.amazon.com/what-is/automated-reasoning/, aws.amazon.com/security/provable-security/.
[2] TPTP has contributed to recognized research in 627 publications that cite [39], according to Google Scholar.

computers. In 2010 a proposal for centralised hardware and software support was submitted to the NSF, and in 2011 a $2.11 million grant[3] was awarded. This grant led to the development and availability of StarExec Iowa [33] in 2012, and a subsequent $1.00 million grant[4] in 2017 expanded StarExec to Miami. StarExec has been central to much progress in logic solvers over the last 10 years, supporting 16 logic solver communities, used for running many annual competitions [1], and supporting many many users. StarExec Iowa provides community infrastructure for many logic solver communities, e.g., ASP, QBF, SAT, SMT, Termination, etc, while StarExec Miami is used by the TPTP community. StarExec Miami has features that take advantage of TPTP standards, and is also used to host CASC.

It was recently announced that StarExec Iowa will be decommissioned. The maintainer of StarExec Iowa explained that "the plan is to operate StarExec as usual for competitions Summer 2024 and Summer 2025, and then put the system into a read-only mode for one year (Summer 2025 to Summer 2026)". The 2017 grant for StarExec Miami paid for the hardware and three years of system administration. The hardware is still hosted by the University of Miami High Performance Computing group, funded on a shoe-string budget by the TPTP World. While StarExec Miami will continue to operate while funding is available, it will not be able to support all the logic solver communities currently using the larger StarExec Iowa. In the long term StarExec will necessarily have to migrate to new compute environments, and several plans are (at the time of writing) being discussed. This paper describes work being done to reengineer StarExec as a cloud-native application using container technology and infrastructure-as-code practices. The first step has been to containerise[5] StarExec and ATP systems so that they can be run on a broad range of computer platforms. The next step in process is to write a new backend in StarExec so that Kubernetes can be used to orchestrate distribution of StarExec job pairs over whatever compute nodes are available. Supported by an Amazon Research Award (see Section 5) a new version of StarExec will be deployed in AWS. This StarExec instance will be fully functional and available to the community (as much as our budget allows). It will also serve as an exemplary implementation for those willing to deploy their own, possibly customized, StarExec on their own computers or in the cloud.

This paper is organized as follows: Section 2 provides a short background to ATP systems, StarExec, and containerisation. Section 3 describes how StarExec has been containerised, and Section 4 describes how ATP systems have been containerised. Section 5 explains how the containerised StarExec and ATP systems will be deployed in a Kubernetes setting. Section 6 concludes and looks forward to future work.

All the software described in the paper is available from ...
`github.com/StarExecMiami/StarExec-ARC`.

2 Background

2.1 StarExec

Figure 1 shows the architecture of the currently deployed StarExec Miami. The hardware consists of a single head node and multiple compute nodes. The head node provides the browser

[3] NSF Awards 1058748 and 1058925, led by Aaron Stump and Cesare Tinelli at the University of Iowa
[4] NSF Award 1730419
[5] Strictly, "images" are built, and the images are deployed in containers. But keeping with common use of the terminology, we say "container images" and "containerise".

interface for users, in particular it accepts job requests that generate job pairs consisting of an ATP system and a problem file, does internal scheduling, and uses the SUN Grid Engine (SGE) to distribute the job pairs to the compute nodes. (For development and testing, the head node can also run job pairs itself using a local backend.) The head node maintains a relational MariaDB database, and all the nodes access an NFS mounted shared file system. The database records everything, including locations of the ATP systems' files and the problem files in the file system. Job pairs executing on a compute node have their time and memory usage limited and reported by the `runsolver` [26] utility (the `BenchExec` [3] utility in StarExec Iowa). The results and resource usage data from completed job pairs are stored in the file system, and recorded in the database. The browser interface provides the necessary facilities for user management, uploading ATP systems, uploading problem files, browsing the ATP systems and problems, creating jobs, imposing resource limits in jobs, tracking job progress, browsing and downloading job results, and deleting ATP systems, problems, jobs, etc.

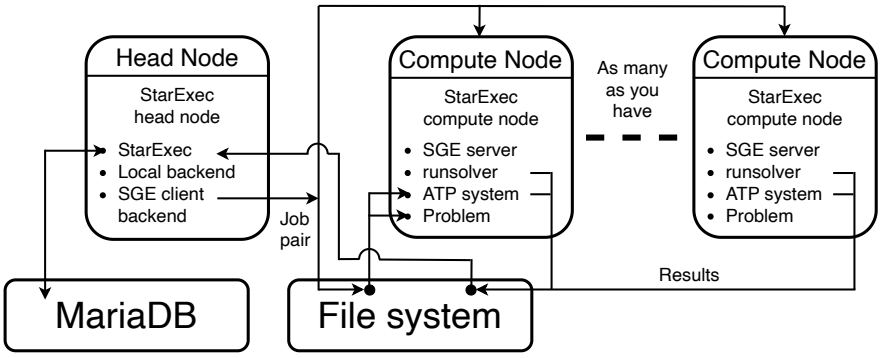

Figure 1: StarExec Architecture

2.2 ATP Systems

ATP Systems are complex pieces of software, typically using advanced data structures [28], sophisticated algorithms [43], and tricky code optimizations [27]. They are written in a variety of programming languages: Prolog [23, 13], Scala [32], C [29], C++ [25], OCaml [17], Python [30], etc. Their build processes include techniques such as parser generators [31], Makefiles, code repositories, specific versions of libraries, etc. For a user who is focussed on an application of ATP, installing an ATP system can be a deal breaker. Many early users selected a weaker system, e.g., Otter [20], for their experiments because it was readily available and easy enough to install. There have been some proposals for standardising the ATP system build process, e.g., `tptp.org/Proposals/SystemBuild.html`, but the diversity of ATP system software makes conformity nigh impossible. An alternative is to push the task back on the system developers, and one approach to this is containerising ATP systems, as discussed in Section 4.

2.3 Containerisation

Containers are a technology stemming from the concept of operating-system-level virtualization.[6] A container is a lightweight, isolated environment that packages and runs applications with all their dependencies as a self-contained unit in user space, while safely sharing the (Linux) kernel with other containers. This encapsulation facilitates seamless software deployment across diverse computing landscapes. Containers are instantiated from read-only *images* that contain all the necessary components and instructions for creating a container, including application code, runtime platform, libraries, environment variables, and configuration files. An image is defined in a file named `Dockerfile` (or `Containerfile`) using a standard syntax. The task of generating a container image definition for an existing application so it can be run as a container is often referred to as "containerisation".

Containerising an application offers numerous benefits, including scalability, resource efficiency, enhanced security, and improved observability. Since containers share the host operating system's kernel, they incur less overhead compared to traditional virtualisation techniques. This characteristic enables containers to be started and stopped quickly, facilitating rapid scaling of applications to meet fluctuating demands. Containerisation also supports observability akin to bare-metal environments through kernel-level mechanisms such as eBPF[7] and cgroups[8], which enable sophisticated monitoring and resource management. The isolation provided by containers helps prevent conflicts between applications and enhances security by limiting the impact of potential vulnerabilities.

Popular containerisation platforms, e.g., Docker, Podman, LXD, and rkt, have significantly contributed to the widespread adoption of container technology within the modern IT landscape. Notably, Kubernetes (often abbreviated as K8s) has emerged as the de-facto industry standard for container orchestration: automating the deployment, scaling, and management of containerised applications. Kubernetes' YAML-based configuration manifests (JSON-variants are also supported) have become widely adopted as a language for declarative infrastructure-as-code (IaC), enabling developers and operations teams to manage infrastructure through declarative, version-controlled code, rather than through the traditional error-prone mixture of imperative scripts and manual processes. IaC thus facilitates consistent, repeatable, and automated provisioning and deployment of servers, networks, and other infrastructure components.

As an "operating system for the cloud", Kubernetes offers several distributions with varying levels of functionality (and complexity). Nowadays, there exist several lightweight production-ready distributions, e.g., k3s (`k3s.io`), k0s (`k0sproject.io`), and microK8s (`microk8s.io`), that greatly simplify the deployment and management of Kubernetes environments, especially in development, testing, and small-scale production scenarios. These distributions provide an accessible entry point for organizations and individuals looking to adopt Kubernetes without the overhead of its full-scale versions, thus democratizing access to this pivotal technology.

3 Containerising StarExec

StarExec (see Section 2.1) is based around a head node that coordinates activities, in particular the creation of jobs as sets of job pairs, with each pair consisting of an ATP system and a problem file. MariaDB is used to store job information and results, and NFS is used to share disk space between the head node and compute nodes. StarExec currently offers two backends for running

[6] See `en.wikipedia.org/wiki/OS-level_virtualization`
[7] Extended Berkeley Packet Filter, see `en.wikipedia.org/wiki/EBPF`
[8] Linux's control groups, see `en.wikipedia.org/wiki/Cgroups`

job pairs: the local backend that runs pairs on the same computer as the head node, and the Sun Grid Engine (SGE) backend that sends pairs out to compute nodes.

So far, the head node with a local backend has been successfully containerised - see the `starexec-containerised` directory of the GitHub repository. It includes ...

- A `Dockerfile` for building a StarExec image with a local backend.
- A StarExec configuration file (database credentials, special StarExec directory paths, default StarExec users, NFS mount path, etc.).
- Various scripts used in the `Dockerfile` to configure and build StarExec. These scripts are responsible for:
  - Installing and configuring StarExec dependencies including Java, Apache Tomcat, ant, MariaDB, SPSS, and more.
  - Creating new user accounts (at the operating system level) used for running jobs.
  - Changing permissions of certain files and directories that StarExec depends on.
  - Building StarExec using ant, which also initializes the database.
- A `README.md` file explaining how to build and run the image.

The deployment of StarExec Miami was a real challenge, requiring installation and configuration of many pieces of software. The containerisation approach aims to make this process simple and repeatable, eliminating the need to understand the complex environment requirements of StarExec. While the containerisation of StarExec with a local backend is somewhat valuable on its own, it is most importantly a first step towards the deployment of a full StarExec cluster in the cloud. Section 5 explains how this will be done.

4 Containerising ATP Systems

While the grand plan is to deploy ATP systems in a containerised StarExec, and in a Kubernetes hosted version of StarExec, containerising ATP systems is independently useful because it allows ATP systems to be easily deployed in users' applications. It would be great if ATP systems developers become super enthusiastic about containerising their systems after reading this section ☺.

The ATP systems' are containerised in a hierarchy, shown in Figure 2. The underlying operating system is `ubuntu:latest` from `dockerhub` ...

`hub.docker.com/_/ubuntu`

The `ubuntu-arc`[9] container image adds to `ubuntu:latest` using `apt-get` to install common software such as `cmake`, `git`, `tcsh`, `python3`, and `wget`. `ubuntu-arc` also creates an `artifacts` directory where the components required for an ATP system's execution are placed.

The `tptp-world` container image provides utilities from the TPTP World that are used by ATP systems, e.g., `SPCForProblem` detects the Specialist Problem Class (SPC) [42] of a problem that is used by some ATP systems to decide on what search parameters to use. To support these utilities some libraries that are not part of the `ubuntu-arc` have to be added. Additionally, the `runsolver` utility for limiting and reporting the resources used by an ATP system is added. (See Section 4.3 for information about the forthcoming `ResourceLimitRun` utility that will replace `runsolver`.) The details of building the *ATP-system:version* and *ATP-system:version*-`RLR` container images are provided in Section 4.1.

[9] "arc" for "Automated Reasoning Containerisation, or Automated Reasoning in the Cloud".

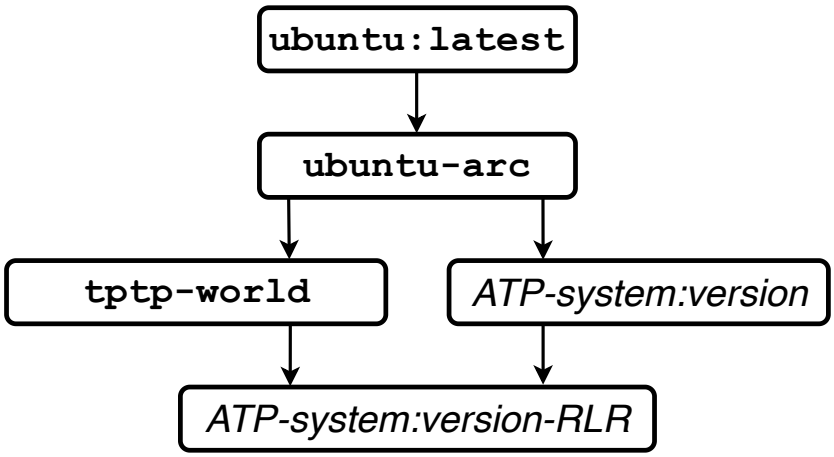

Figure 2: ATP System Container Image Hierarchy

4.1 Building ATP System Containers

Each *ATP-system:version* container image is built on top of the `ubuntu-arc` container image, and with the `tptp-world` container image forms the base for the final *ATP-system:version*-`RLR` container image. The *ATP-system* is the container name, and the *version*/*version*-`RLR` are the container tags. Podman[10] requires the container name to be lowercase, so, e.g., E's container is named `eprover`. The "`RLR`" refers to the "Resource Limited Run" program used to monitor and limit the resources used by the ATP system, either `runsolver` or `ResourceLimitRun`. The files for containerising some ATP systems are in the `provers-containerised` directory of the GitHub repository. A `Makefile` to containerise E, Leo-III, and Vampire is included.

Each *ATP-system:version* container image adds the ATP system's executables to `ubuntu-arc`. The ATP system is retrieved online, e.g., from a GitHub repository, and the necessary commands to build the executables are run. The executables are copied into the `/artifacts` directory. The choice of which version of the ATP system to containerise is made inside the `Dockerfile`. This localization is necessary because the processes for retrieving and building particular ATP system versions vary from system to system and from version to version. An *ATP-system:version* container image must include a `run_system` script to run the ATP system, using whatever incantations are necessary. The parameters for running the ATP system are provided to the `run_system` script in "`RLR`" environment variables (see Section 4.2). Appendix A shows E's `run_system` script. It invokes the `eprover` or `eprover-ho` binary, depending on whether the problem is first-order or higher-order. Depending on the intent, the appropriate command line arguments are given to the selected binary along with the problem file and time limit. Figure 3 shows the `Dockerfile` used to create E's `eprover:3.0.03` container image, using the command "`podman build -t eprover:3.0.03 .`".

Each *ATP-system:version*-`RLR` container image is based on its *ATP-system:version* container image and the `tptp-world` container image. *ATP-system:version*-`RLR` primarily extends `tptp-world`, and copies over only what is necessary from *ATP-system:version*. This simple arrangement allows a generic `Dockerfile` to be used, parameterised by the under-

[10] Our containerisation efforts are carried out using Podman, which is designed to work as a drop-in replacement for Docker (simply aliasing `podman` to `docker` is endorsed in the documentation).

```
#-------------------------------------------------------------
FROM ubuntu-arc

# Clones repository
ARG E_VERSION=E-3.0.03
RUN git clone --depth 1 --branch $E_VERSION https://github.com/eprover/eprover.git

# Set working directory to cloned sources directory
WORKDIR /eprover

# Builds first-order executable
RUN ./configure --bindir=/artifacts && \
    make && \
    make install

# Builds higher-order executable
RUN ./configure --enable-ho && \
    make rebuild
RUN cp PROVER/eprover-ho /artifacts/eprover-ho

# run_system script
ADD run_system /artifacts/
#-------------------------------------------------------------
```

Figure 3: The `Dockerfile` for E's `-build`

lying *ATP-system:version*. The `ENTRYPOINT` in *ATP-system:version*-`RLR` is the `runsolver` utility from `tptp-world`, which is used to run the ATP system (see Section 4.2). Figure 4 shows the `Dockerfile` to create E's `eprover:3.0.03-RLR` container image, using the command "`podman build -t eprover:3.0.03 RLR --build-arg PROVER_IMAGE=eprover:3.0.03 .`". The *ATP-system:version*-`RLR` container images are pushed to `dockerhub` in ...

hub.docker.com/repositories/tptpstarexec

which has a directory for each ATP system. The pushed container images are tagged as *ATP-system-name*:*ATP-system-version*-`RLR`-*architecture*, where *architecture* is, e.g., `arm64` or `amd64`.

4.2 Running *ATP-system:version*-RLR Containers

An *ATP-system:version*-`RLR` container image is started using `podman run`. The parameters for running the ATP system are passed into the container in environment variables, using the `-e` option: `RLR_INPUT_FILE` provides the problem file name, `RLR_CPU_LIMIT` provides the CPU time limit in seconds (0 by default, to indicate no limit), `RLR_WC_LIMIT` provides the wall clock time limit in seconds (0 by default, to indicate no limit), `RLR_MEM_LIMIT` provides the memory limit in MiB (0 by default, to indicate no limit), and `RLR_INTENT` indicates the user's intent[11] (`THM` by default). The problem file is passed into the running container using the `-v` option to mount the directory containing the problem file to a directory inside the container,

[11] An *intent* is a tag such as `THM` or `SAT`, indicating that the ATP system should try to prove (or, equivalently for most systems, show that the problem is unsatisfiable) or disprove (or, equivalently for most systems, show that the problem is satisfiable) the conjecture, respectively.

```
#-------------------------------------------------------------
ARG PROVER_IMAGE

FROM ${PROVER_IMAGE} AS builder
FROM tptp-world

ENV PATH=".:${PATH}"
WORKDIR /artifacts

# System specific stuff
COPY --from=builder /artifacts/* /artifacts/

ENTRYPOINT ["runsolver"]
#-------------------------------------------------------------
```

Figure 4: The generic `Dockerfile` for building `-RLR` container images

and setting the `RLR_INPUT_FILE` environment variable to the name of the problem file in the directory inside the container. The command line parameters for `runsolver` (the `ENTRYPOINT` in the *ATP-system:version*`-RLR` container image) and `run_system` are provided as the remaining parameters to `podman run`. For example, to run the `eprover:3.0.03-RLR` container image on the problem `MGT019+2.p`, the `podman run` could be ...

```
    podman run eprover:3.0.03-RLR -v .:/artifacts/CWD
-e RLR_INPUT_FILE='/artifacts/CWD/MGT019+2.p' -e RLR_CPU_LIMIT='60'
-e RLR_WC_LIMIT='60' -e RLR_MEM_LIMIT='0' -e RLR_INTENT='SAT'
--timestamp -C 60 -W 60 run_system
```

The "`--timestamp -C 60 -W 60`" are command line parameters to `runsolver`.

A Python script `run_image.py` is provided to simplify and standardize running *ATP-system:version*`-RLR` container images. The script is shown in Appendix B. The script must have an *ATP-system:version*`-RLR` container image name as a command line argument. By default `run_image.py` runs the *ATP-system:version*`-RLR` with the problem taken from `stdin`, imposing no CPU, wall clock, or memory limits, with the `THM` intent. All the parameters can be changed with further command line options.

4.3 The ResourceLimitedRun Utility

When the TPTP World's SystemOnTPTP service [34] was first made available [35] it used a Perl program called `TreeLimitedRun` to monitor and limit ATP systems' use of CPU time, wall clock time, and memory. As the name suggests, the principle was to monitor the forest of process hierarchies started by an ATP system, understanding that some of the processes might be orphaned and adopted by the `init` process (now `systemd` and others). `TreeLimitedRun` was superseded by `runsolver` [26] that is written in C++, and adopted the same principle for monitoring processes. More recently, `BenchExec` [3], which is used in StarExec Iowa, has taken advantage of Linux's cgroup v2 subsystem, which provides operating system level support for monitoring processes. `BenchExec` is written in Python, with rather heavy installation requirements. The new `ResourceLimitedRun` utility is written in C, and also uses Linux's cgroup v2 subsystem. `ResourceLimitedRun` has the same command line parameters as `runsolver`, and thus can be substituted for `runsolver` (and `BenchExec`). `ResourceLimitedRun` is being tested at the time of writing, and hopefully will be deployed at the time of presentation!

5 Towards a Cloud-native StarExec

The term "cloud-native" has increasingly become synonymous with an approach to designing and operating applications that fully leverage the benefits of the cloud computing model.[12] Cloud-native applications are distinguished by their ability to scale effectively, utilising the cloud's capability to dynamically allocate resources. This development paradigm is closely aligned with DevOps practices that emphasize collaboration between development and operations teams to automate the process of software delivery and infrastructure changes. It inherently supports infrastructure-as-code (IaC), a key DevOps practice, enabling the management and provisioning of infrastructure through declarative, version-controlled definition files that are both human- and machine-readable.

Containers (see Section 2.3) play a pivotal role in cloud-native development. `Dockerfile`s are used to specify the steps to create a container image, embodying the IaC philosophy by detailing the desired (partial) state of a containerised application. Similarly, Kubernetes YAML manifests define, in a declarative fashion, how application components are deployed and run on Kubernetes clusters, aligned with the IaC paradigm.

The synthesis of these practices allows for the entire stack, from infrastructure to application, to be declaratively specified, versioned, and automatically deployed as required. The reliance on mainstream open source technologies such as the CNCF Kubernetes and Podman projects (`www.cncf.io/projects`) offers unparalleled flexibility, scalability, and portability, free from the constraints of single vendors or platforms. An open distribution model ensures that StarExec's infrastructure "as code" is readily accessible for modification and distribution, e.g., by cloning or forking from our GitHub repository. Adopting these technologies will allow ATP systems and StarExec, including the requisite infrastructure, to be deployed by ATP system developers and users in their preferred cloud environment or even in on-premises servers. This approach significantly simplifies the process of utilizing state-of-the-art ATP technology, making it much more easily usable by anyone, anywhere.

5.1 Re-engineering StarExec for the Cloud

Recalling the current architecture of StarExec described in Section 3, several areas for improvement have been identified to better serve the needs for re-engineering. Our ongoing efforts include:

1. **Utilization of containerised ATP systems** (see Section 4), which will be hosted in a publicly accessible container image registry, instead of the current approach of requiring StarExec users to build and upload a StarExec `.tgz` package according to StarExec specifications.

2. **Adding an abstraction layer for database communication** with the relational database (currently MariaDB) used to persist job information. This layer will allow the database component to operate in its own container, significantly reducing coupling. Furthermore, by eliminating MariaDB-specific bindings, compatibility with other database systems will be enabled. This flexibility allows for seamless integration with existing SQL databases within the user's infrastructure, enhancing portability and adaptability.

[12] For more information refer to the initiatives led by the Cloud Native Computing Foundation (CNCF) at `www.cncf.io`, which advocates for the adoption of this paradigm by fostering and sustaining an ecosystem of open source, vendor-neutral projects.

3. **Using Kubernetes job scheduling facilities**, thereby completely replacing the current SGE cluster management. This change offers numerous benefits:
    - **Scalability:** Kubernetes excels at managing and scaling containerised applications, adapting to fluctuating workloads with ease. It also seamlessly integrates with most infrastructure provisioning tools, supporting both cloud and on-premise platforms.
    - **Monitoring:** A vast array of observability tools (encompassing logging, metrics, tracing, etc.) support seamless integration with Kubernetes. Additionally, with its self-healing features, Kubernetes can automatically restart failed containers, replace and reschedule containers when nodes die, and kill non-responsive containers.
    - **Efficiency:** Similar to SGE and other cluster management software such as Slurm and Torque, Kubernetes optimizes the use of underlying hardware by efficiently scheduling jobs and managing resources.[13]
    - **Flexibility:** Kubernetes' extensible architecture allows for custom schedulers and automated scaling decisions, enabling it to support a wide range of workloads, including stateless, stateful, and batch processing.

Figure 5 shows a generic architecture of the future re-engineered StarExec using Kubernetes.

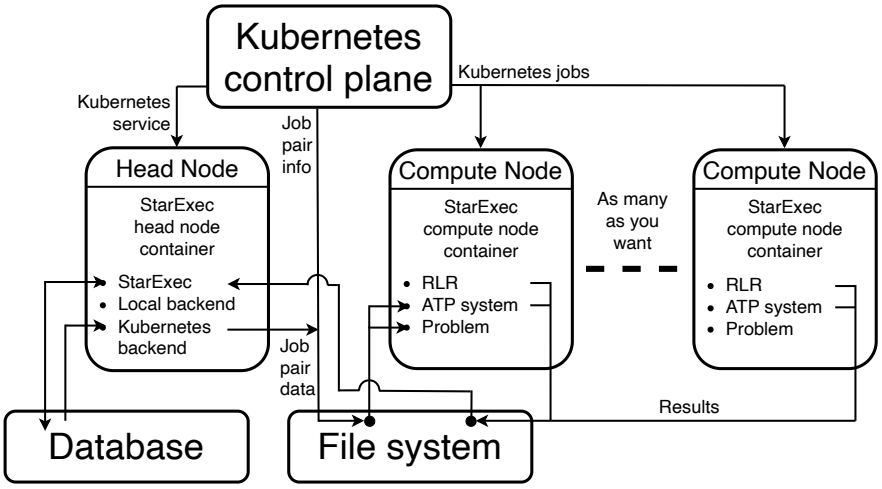

Figure 5: Projected StarExec generic architecture

5.2 StarExec in AWS

An Amazon Research Award[14] has been granted to deploy StarExec in AWS. This will not only help fund the development efforts discussed in Section 5.1, but will also fund a first fully-

[13] Certainly, Kubernetes does not outperform traditional High-Performance Computing (HPC) software within their specialized application domains. Kubernetes' extensible architecture facilitates interfacing with HPC systems through custom schedulers if the need arises (see `kubernetes.io/docs/concepts/extend-kubernetes`). In this setup, Kubernetes oversees container orchestration, while delegating the scheduling of intensive computing tasks to specialized HPC software.

[14] Amazon Research Award, Fall 2023. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors, and do not reflect the views of Amazon.

functional reference deployment of StarExec in the AWS cloud. The generic architecture in Figure 5 will be instantiated using concrete AWS-managed services, as shown in Figure 6.

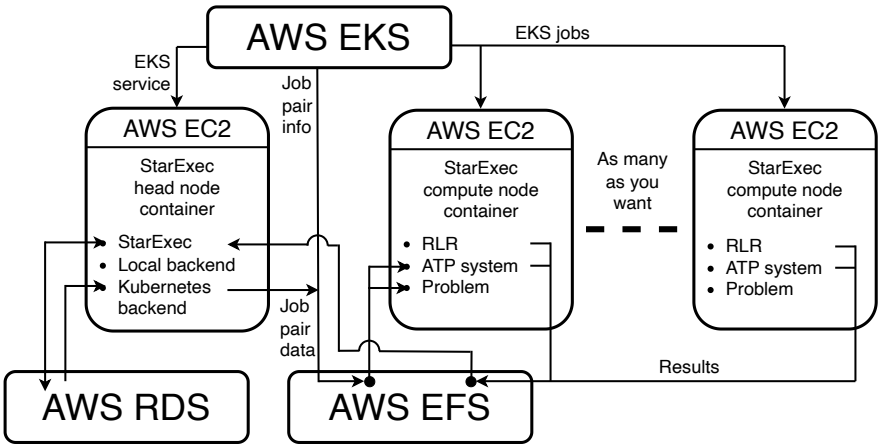

Figure 6: Architecture in AWS

- The Kubernetes control plane will be managed by AWS Elastic Kubernetes Service (EKS).
- The StarExec head and compute nodes will run on suitable Amazon EC2 instances, currently planned to be `x2iedn.xlarge` instances that have four Intel Xeon Scalable vCPUs running up to 3.5GHz, and 128 GiB memory.
- The database will be Amazon Relational Database (RDS).
- The file system will be Amazon Elastic File System (EFS).
- The ATP systems' containerisation can be made compatible with (possibly be exactly) the Amazon Trusted Solver format, as was recently used in the SMT and SAT competitions[15].

Leveraging AWS-managed services will expedite the delivery of StarExec's initial cloud-native version to the community. This approach will particularly benefit teams planning to deploy StarExec on their own AWS accounts or through AWS grants. The initial release will be rigorously tested through the migration of the TPTP community from StarExec Miami to the new StarExec AWS platform. We are particularly enthusiastic about collaborating with teams interested in deploying StarExec on their on-premise infrastructure or within university HPC clusters.

6 Conclusion

This paper has described work being done to containerise StarExec and ATP systems so that they can be run on a broad range of computer platforms. Additionally, this work explains plans to build backend in StarExec so that Kubernetes can be used to orchestration distribute of StarExec job pairs over whatever compute nodes are available.

This is ongoing work – some of the work is still in progress, particularly embedding StarExec in Kubernetes on AWS. Hopefully the future will include StarExec being flexibly available in online compute clusters.

[15] github.com/aws-samples/aws-batch-comp-infrastructure-sample

References

[1] E. Bartocci, D. Beyer, P.E. Black, G. Fedyukovich, H. Garavel, A. Hartmanns, M. Huisman, F. Kordon, J. Nagele, M. Sighireanu, B. Steffen, M. Suda, G. Sutcliffe, T. Weber, and A. Tamada. TOOLympics 2019: An Overview of Competitions in Formal Methods. In T. Vojnar and L. Zhang, editors, *Proceedings of the 2019 International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 11429 in Lecture Notes in Computer Science, page To appear. Springer-Verlag, 2019.

[2] C. Benzmüller and B. Woltzenlogel Paleo. Automating Gödel's Ontological Proof of God's Existence with Higher-order Automated Theorem Provers. In T. Schaub, editor, *Proceedings of the 21st European Conference on Artificial Intelligence*, pages 93–98, 2014.

[3] D. Beyer, S. Löwe, and P. Wendler. Reliable Benchmarking: Requirements and Solutions. *International Journal on Software Tools for Technology Transfer*, 21:1–29, 2019.

[4] A. Bruni, E. Drewsen, and C. Schürmann. Towards a Mechanized Proof of Selene Receipt-Freeness and Vote-Privacy. In R. Krimmer, M. Volkamer, N. Braun Binder, N. Kersting, O. Pereira, and C. Schürmann, editors, *Proceedings of the International Joint Conference on Electronic Voting, E-Vote-ID 2017*, number 10615 in Lecture Notes in Computer Science, pages 110–126. Springer-Verlag, 2017.

[5] M. Caminati, M. Kerber, C. Lange, and C. Rowat. Sound Auction Specification and Implementation. In M. Feldman, M. Schwarz, and T. Roughgarden, editors, *Proceedings of the 16th ACM Conference on Economics and Computation*, pages 547–564. ACM Press, 2015.

[6] V. Chaudri, B. Cheng, A. Overholtzer, J. Roschelle, A. Spaulding, P. Clark, M. Greaves, and D. Gunning. Inquire Biology: A Textbook that Answers Questions. *AI Magazine*, 34(3), 2013.

[7] D. Cok, A. Stump, and T. Weber. The 2013 Evaluation of SMT-COMP and SMT-LIB. *Journal of Automated Reasoning*, 55(1):61–90, 2015.

[8] B. Cook. Formal Reasoning About the Security of Amazon Web Services. In H. Chockler and G. Weissenbacher, editors, *Proceedings of the 30th International Conference on Computer Aided Verification*, number 10981 in Lecture Notes in Computer Science, pages 38–47. Springer-Verlag, 2018.

[9] L. Dennis, M. Fisher, M. Slavkovik, and M. Webster. Formal Verification of Ethical Choices in Autonomous Systems. *Robotics and Autonomous Systems*, 77:1–14, 2016.

[10] R. Hähnle and M. Huisman. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. In B. Steffen and G. Woeginger, editors, *Computing and Software Science: State of the Art and Perspectives*, number 10000 in Lecture Notes in Computer Science, pages 345–373. Springer-Verlag, 2019.

[11] M.T. Hannan. Rethinking Age Dependence in Organizational Mortality: Logical Formalizations. *American Journal of Sociology*, 104:126–164, 1998.

[12] J. Harrison. Floating-Point Verification using Theorem Proving. In M. Bernardo and A. Cimatti, editors, *Proceedings of the 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, number 3965 in Lecture Notes in Computer Science, pages 211–242. Springer-Verlag, 2006.

[13] S. Holden. Connect++: A New Automated Theorem Prover Based on the Connection Calculus. In J. Otten and W. Bibel, editors, *Proceedings of the 1st International Workshop on Automated Reasoning with Connection Calculi*, number 3613 in CEUR Workshop Proceedings, pages 95–106, 2023.

[14] A. Hommersom, P. Lucas, and P. van Bommel. Automated Theorem Proving for Quality-checking Medical Guidelines. In G. Sutcliffe, B. Fischer, and S. Schulz, editors, *Proceedings of the Workshop on Empirically Successful Classical Automated Reasoning*, 2005.

[15] H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. In I. Gent, H. van Maaren, and T. Walsh, editors, *Proceedings of the 3rd Workshop on the Satisfiability Problem*,

pages 283–292. IOS Press, 2000.

[16] J. Horner. A Computationally Assisted Reconstruction of an Ontological Argument in Spinoza's The Ethics. *Open Philosophy*, 2:219–229, 2019.

[17] K. Korovin. Implementing an Instantiation-based Theorem Prover for First-order Logic. In C. Benzmüller, B. Fischer, and G. Sutcliffe, editors, *Proceedings of the 6th International Workshop on the Implementation of Logics*, number 212 in CEUR Workshop Proceedings, pages 63–63, 2006.

[18] T. Libal. Towards Automated GDPR Compliance Checking. In F. Heintz, M. Milano, and B. O'Sullivan, editors, *Proceedings of the International Workshop on the Foundations of Trustworthy AI Integrating Learning, Optimization and Reasoning*, number 12641 in Lecture Notes in Computer Science, pages 3–19, 2020.

[19] C. Marché and H. Zantema. The Termination Competition. In F. Baader, editor, *Proceedings of the 18th International Conference on Term Rewriting and Applications*, number 4533 in Lecture Notes in Computer Science, pages 303–313, 2007.

[20] W.W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA, 2003.

[21] T. Nipkow. Social Choice Theory in HOL: Arrow and Gibbard-Satterthwaite. *Journal of Automated Reasoning*, 43(3):289–304, 2009.

[22] P. Oppenheimer and E. Zalta. A Computationally-Discovered Simplification of the Ontological Argument. *Australasian Journal of Philosophy*, 89(2):333–349, 2011.

[23] J. Otten. 20 Years of leanCoP - An Overview of the Provers. In J. Otten and W. Bibel, editors, *Proceedings of the 1st International Workshop on Automated Reasoning with Connection Calculi*, number 3613 in CEUR Workshop Proceedings, pages 4–22, 2023.

[24] H. Prakken and G. Sartor. Law and Logic: A Review from an Argumentation Perspective. *Artificial Intelligence*, 227:214–245, 2015.

[25] A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.

[26] O. Roussel. Controlling a Solver Execution with the `runsolver` Tool. *Journal of Satisfiability, Boolean Modeling and Computation*, 7(4):139–144, 2011.

[27] S. Schulz. Algorithms and Data Structures for First-Order Equational Deduction. In C. Benzmüller, B. Fischer, and G. Sutcliffe, editors, *Proceedings of the 6th International Workshop on the Implementation of Logics*, number 212 in CEUR Workshop Proceedings, pages 1–6, 2006.

[28] S. Schulz. Simple and Efficient Clause Subsumption with Feature Vector Indexing. In M.P. Bonacina and M. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, number 7788 in Lecture Notes in Artificial Intelligence, pages 45–67. Springer-Verlag, 2013.

[29] S. Schulz, S. Cruanes, and P. Vukmirović. Faster, Higher, Stronger: E 2.3. In P. Fontaine, editor, *Proceedings of the 27th International Conference on Automated Deduction*, number 11716 in Lecture Notes in Computer Science, pages 495–507. Springer-Verlag, 2019.

[30] S. Schulz and A. Pease. Teaching Automated Theorem Proving by Example: PyRes 1.2 (system description). In N. Peltier and V. Sofronie-Stokkermans, editors, *Proceedings of the 10th International Joint Conference on Automated Reasoning*, number 12167 in Lecture Notes in Computer Science, pages 158–166, 2020.

[31] A. Steen. Scala TPTP Parser v1.5, 2021. DOI: 10.5281/zenodo.5578872.

[32] A. Steen and C. Benzmüller. The Higher-Order Prover Leo-III. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *Proceedings of the 8th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Artificial Intelligence, pages 108–116, 2018.

[33] A. Stump, G. Sutcliffe, and C. Tinelli. StarExec: a Cross-Community Infrastructure for Logic Solving. In S. Demri, D. Kapur, and C. Weidenbach, editors, *Proceedings of the 7th International Joint Conference on Automated Reasoning*, number 8562 in Lecture Notes in Artificial Intelligence,

pages 367–373, 2014.

[34] G. Sutcliffe. SystemOnTPTP. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Intelligence, pages 406–410. Springer-Verlag, 2000.

[35] G. Sutcliffe. TPTP, TSTP, CASC, etc. In V. Diekert, M. Volkov, and A. Voronkov, editors, *Proceedings of the 2nd International Symposium on Computer Science in Russia*, number 4649 in Lecture Notes in Computer Science, pages 6–22. Springer-Verlag, 2007.

[36] G. Sutcliffe. The SZS Ontologies for Automated Reasoning Software. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics*, number 418 in CEUR Workshop Proceedings, pages 38–49, 2008.

[37] G. Sutcliffe. The TPTP World - Infrastructure for Automated Reasoning. In E. Clarke and A. Voronkov, editors, *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 6355 in Lecture Notes in Artificial Intelligence, pages 1–12. Springer-Verlag, 2010.

[38] G. Sutcliffe. The CADE ATP System Competition - CASC. *AI Magazine*, 37(2):99–101, 2016.

[39] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[40] G. Sutcliffe. Stepping Stones in the TPTP World. In C. Benzmüller, M. Heule, and R. Schmidt, editors, *Proceedings of the 12th International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, page Invited paper, 2024.

[41] G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, number 4130 in Lecture Notes in Artificial Intelligence, pages 67–81. Springer, 2006.

[42] G. Sutcliffe and C.B. Suttner. Evaluating General Purpose Automated Theorem Proving Systems. *Artificial Intelligence*, 131(1-2):39–54, 2001.

[43] A' Voronkov. Algorithms, Datastructures, and Other Issues in Efficient Automated Deduction. In R. Gore, A. Leitsch, and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning*, number 2083 in Lecture Notes in Artificial Intelligence, pages 13–28. Springer-Verlag, 2001.

[44] M. Yadav. On the Synthesis of Machine Learning and Automated Reasoning for an Artificial Synthetic Organic Chemist. *New Journal of Chemistry*, 41(4):1411–1416, 2017.

A E's run_system script

```
#------------------------------------------------------------------------------
#!/bin/tcsh

setenv HERE `dirname $0`
setenv TEMPDIR `mktemp -d`
setenv PROBLEMFILE $TEMPDIR/E---3.1_$$.p
onintr cleanup

#----Add extra ()s for THF and TXF
$HERE/tptp4X -t uniquenames4 -x $RLR_INPUT_FILE > $PROBLEMFILE

set SPCLine=`grep -E "^% SPC " $PROBLEMFILE`
if ("$SPCLine" != "") then
    set ProblemSPC = `expr "$SPCLine" : "^% SPC  *: *\([^ ]*\)"`
else
    set ProblemSPC = `$HERE/SPCForProblem $RLR_INPUT_FILE`
endif
set Mode = $RLR_INTENT

set CommonParameters="--delete-bad-limit=2000000000 --definitional-cnf=24 \
-s --print-statistics -R --print-version --proof-object --cpu-limit=$RLR_WC_LIMIT"
if ("$Mode" == "THM") then
    if (`expr "$ProblemSPC" : "TH0_.*"`) then
        echo "Running higher-order theorem proving"
        $HERE/eprover-ho $CommonParameters --auto-schedule=8 $PROBLEMFILE
    else
        echo "Running first-order theorem proving"
        $HERE/eprover $CommonParameters --auto-schedule=8 $PROBLEMFILE
    endif
else
    echo "Running first-order model finding"
    $HERE/eprover $CommonParameters --satauto-schedule=8 $PROBLEMFILE
endif

cleanup:
    echo "% E exiting"
    rm -rf $TEMPDIR
#------------------------------------------------------------------------------
```

B run_image.py

```
#------------------------------------------------------------------------------
#!/usr/bin/env python3

import argparse
import subprocess
import os, sys
import shutil

def getRLRArgs(args):
    mem_part = f" -M {args.memory_limit}" if args.memory_limit > 0 else ""
    return "--timestamp --watcher-data /dev/null -C " + \
f"{args.cpu_limit} -W {args.wall_clock_limit}{mem_part}"

def getEnvVars(args):
    return " ".join([f"-e {k}='{v}'" for k, v in [
        ("RLR_INPUT_FILE", "/artifacts/CWD/problemfile"),
        ("RLR_CPU_LIMIT", args.cpu_limit), ("RLR_WC_LIMIT", args.wall_clock_limit),
        ("RLR_MEM_LIMIT", args.memory_limit), ("RLR_INTENT", args.intent),
    ]])

def makeBenchmark(problem):
    if problem:
        shutil.copy(problem, "./problemfile")
    else:
        with open('./problemfile', 'w') as problemfile:
            problemfile.write(sys.stdin.read())

if __name__ == "__main__":
    parser = argparse.ArgumentParser("Wrapper for a podman call to a prover image")
    parser.add_argument("image_name",
help="Image name, e.g., eprover:3.0.03-RLR-arm64")
    parser.add_argument("-P", "--problem",help="Problem file if not stdin")
    parser.add_argument("-C", "--cpu-limit", default=0, type=int,
help="CPU time limit in seconds, default=none")
    parser.add_argument("-W", "--wall-clock-limit", default=0, type=int,
help="Wall clock time limit in seconds, default=none")
    parser.add_argument("-M", "--memory-limit", default=0, type=int,
help="Memory limit in MiB, default=none")
    parser.add_argument("-I", "--intent", default="THM", choices=["THM", "SAT"],
help="Intention (THM, SAT, etc), default=THM")
    args = parser.parse_args()
    if args.wall_clock_limit == 0 and args.cpu_limit != 0:
        args.wall_clock_limit = args.cpu_limit

    command = f"podman run {getEnvVars(args)} -v .:/artifacts/CWD -t " + \
f"{args.image_name} {getRLRArgs(args)} run_system"
    makeBenchmark(args.problem)
    subprocess.run(command, shell=True)
    os.remove("./problemfile")
#------------------------------------------------------------------------------
```

Shared Terms and Cached Rewriting

Stephan Schulz

DHBW Stuttgart
Stuttgart, Germany
schulz@eprover.org

Abstract

We describe the implementation of first-order terms, the central data structure of most modern automated theorem provers, as perfectly shared immutable term DAGs in E. We demonstrate typical gains possible with this structure (reducing the number of term nodes typically by orders of magnitude) and discuss some of the side benefits of such a representation. One of these benefits is the ability to easily implement cached rewriting, improving the performance of rewriting-based simplification. We discuss lessons learned and some potential future work.

1 Introduction

Shared terms seem to have become a staple among high-performance automated theorem provers, but this is rarely mentioned outside the source code. In particular, it is hard to find descriptions of their implementation and performance. In this paper, we try to alleviate this difficulty and describe the implementation of shared terms and cached rewriting in E [Sch02, SCV19]. E is a mature theorem prover, written in ANSI C, and continually developed for about a quarter of a century.

First-order terms, such as $f(X, a)$ or $f(g(g(a)), f(X, b))$, are the most central element of most automated first-order theorem prover. Their implementation is probably the most critical data structure in particular for saturating systems, which generate new terms in prodigious numbers during proof search. Such systems, like e.g. E, Vampire [KV13], SPASS [WDF+09], Prover9 [McC10] and Twee [Sma21] have dominated the field of automated theorem proving for the last decades. They are typically based on variants of the superposition calculus [BG94] (or its unit-equational counterpart, unfailing completion [BDP89]), employ resolution [Rob65] and/or superposition (an ordering-constrained form of paramodulation [RW69]) as the main inference rules to create new clauses, and rewriting (sometimes called *demodulation*) and subsumption as the major mechanisms to simplify and remove clauses.

There are several different ways to implement terms, from simple trees as e.g. in the completion-based prover DISCOUNT [DKS97] and many early provers, to flat terms [Chr93] or string terms as used in Waldmeister [LH02]. When we started the development of E, one of the core ideas was to structure the prover around *shared terms*, i.e. a term structure in which every term (and subterm) was represented only once, and different occurrences of the same term simply point to the one copy stored in a *term bank*. Such a term bank represents a forest of term trees as a single directed acyclic graph (DAG).

The first implementation of this idea in E was realised as a *dynamic* term bank with mutable terms. Rewriting, one of the core simplification techniques, would actively change terms in the term bank. Changes were propagated to superterms, possibly leading to large, non-local changes as the result of a single rewrite step. Memory management of term cells was handled via reference-counting garbage collection. We did a comparative evaluation of shared and unshared (flat) terms by comparing the performance of E and Waldmeister, both tuned to behave as similar as possible [LS01]. The result was somewhat disappointing - while shared

terms represented the proof state using much fewer term cells, the propagation of rewriting to superterms nearly exactly cancelled out the benefits gained by rewriting each subterm at most once.

Since the dynamic term bank implementation did not result in performance benefits, but significantly complicated overall system design and in particular proof reconstruction, we changed the implementation. The new version uses the same basic DAG structure, but terms themselves are now *immutable*. Rewriting of terms in clauses is always triggered from the clause level (so no complex notification or bookkeeping of changes is needed). To speed up rewriting, we cache the result of rewrite steps, i.e. we add an annotated link to a rewritten term, pointing the resulting term and giving a justification for the rewrite (normally the clause that was used to perform the rewrite). Term cell memory is still handled by garbage collection, but now using a mark-and-sweep garbage collector that is only triggered at strategic locations in the code (e.g. after axiom selection and clause normal form transformation), or if there is active memory pressure.

In this paper, we describe this second implementation for the first time in some detail, and we report on some experiences and measurements. In an ideal world, theorem provers would use an abstract interface to all major data types, and it would be possible to just plug different data structures in to get perfect performance comparisons. However, despite some attempts this has never been achieved for high-performance theorem provers. This is especially true for the term data type, for two reasons: First, the term data type is so central that its design imposes significant constraints on overall system design and architecture. And secondly, theorem proving has been (and is) an ongoing research field, and new ideas often require new methods for accessing and manipulating terms. However, we believe that the statistics we present below provide some insight into the value of shared terms and cached rewriting.

1.1 Background

We assume that the reader is familiar with the basic design of modern saturating theorem provers. The proof state is represented by a set of clauses, where each clause is a disjunctively interpreted multi-set of literals and each literal is a signed atom - in the case of E either an equation or a disequation between terms. New clauses are created by generating inference rules, mostly based on unification, with most clauses generated by *superposition* and/or *resolution*. The proof state is reduced using simplification rules, often based on matching - in particular *subsumption* and *rewriting* or *demodulation* with unit clauses. Provers based on the *given-clause loop* split the proof state into a set of *processed* or *active* clauses, which is interreduced, and a set of *unprocessed* or *passive* clauses which may be partially simplified, but have not yet participated in generating inferences. The most important search decision is the selection of the next of these unprocessed clauses for processing.

2 Term Banks and Shared Terms

In the following, we assume a first-order signature $F = \{f_1/a_1, f_2/a_2, \ldots\}$ of function symbols with associated arities, and a set of variables V. In practice, F is always finite, but may grow over time, e.g. by introducing Skolem symbols or names for definitions. For our purposes, we don't need to distinguish (proper) function symbols and predicate symbols. The set V of variables is conceptually countably infinite, but we only ever need a finite subset.

In E, and in many other theorem provers, function symbols are encoded as small positive integers, which serve as indices into a table representing the full signature, including externally

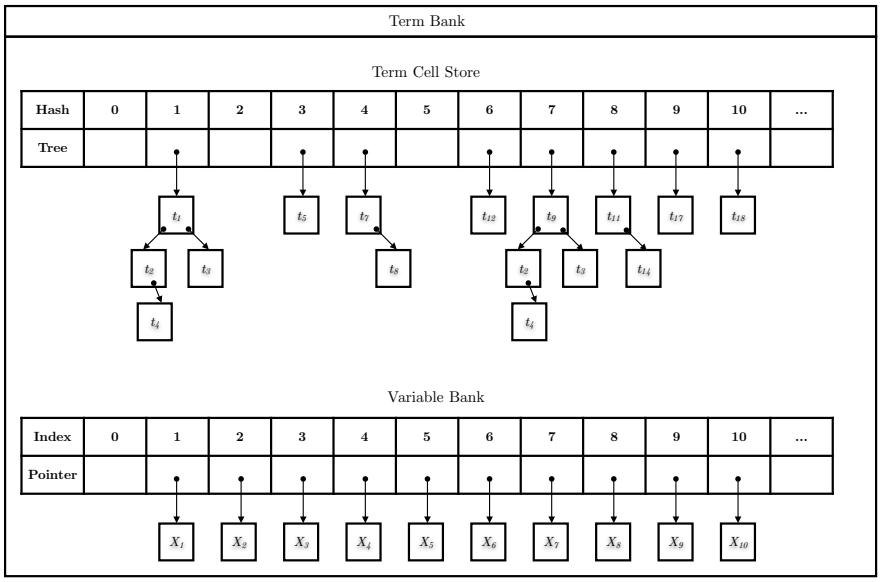

Figure 1: Term bank architecture

visible names of function symbols and meta-properties such as arities. Variables are encoded as small negative integers, with a mapping from input variable names to these integers provided by temporary translation tables during parsing. Since variables are necessarily renamed frequently during proof search, their names in the input are not typically maintained long-term.

Terms are either variables, or they are constructed from existing terms $t_1, \ldots, t_n$ and a function symbol $f/n \in F$, yielding $f(t_1, \ldots, t_n)$. Notice that for a symbol $c/0 \in F$ (a *constant*), $c()$ is a term. In this case, we usually omit the parentheses.

2.1 Basic implementation

A term bank is a data structure that stores terms and allows reasonably efficient access to terms. In E, terms are represented by pointers to *term cells*, which are essentially homogeneous. A term cell contains an encoding of the function symbol or variable (called the `f_code`), the arity of the term, a set of invariant properties (see the next section), and a dynamic length array of pointers to subterms. As per the above definition, a term is identified by its `f_code` and the list of argument terms. These make up the key under which a term can be found in the term bank.

The main function of the term bank is to return a pointer to a shared term syntactically identical to an arbitrary term handed to it, in other words: to convert (potentially) unshared terms into shared terms, or to find the unique existing equivalent of a given term.

In the definition above we have distinguished two different kinds of terms: Variables, and composite terms starting with a function symbol. While we represent both of these types using standard term cells, they are stored differently. Variables are stored in the *variable bank* of a term bank. The variable bank is a dynamic array of pointers to (variable) term cells, indexed by the (negated) `f_code` of the variable. Thus, finding a shared variable can be done in $\Theta(1)$. Variable cells are not garbage collected, since they are reused over and over again, and are quite low in numbers.

Composite terms, on the other hand, are stored in a *term cell store* data structure. This is implemented as a large hash table with collisions resolved externally via splay trees [ST85] (a self-adjusting variant of binary search trees). Composite terms of the form $f(t_1, \ldots, t_n)$ are inserted/found bottom up. First, we compute pointers $s_1, \ldots, s_n$ to shared versions of the argument terms $t_1, \ldots, t_n$ (note that this may involve further recursion). We then consider the sequence `f_code,` s_1`, ...,` s_n as the search key for the term in the term bank. We compute a hash code from the `f_code` and up to two argument pointers by xoring their (shifted) binary representations and masking them to 15 bits, selecting one of 32768 possible term cell trees. Using at most two argument pointers simplifies the hash computation and is sufficient to give a relatively even distribution of terms over hash values. Since the number of terms is much larger than the hash table, conflicts are unavoidable, and are resolved by storing not terms, but term sets (represented by splay trees) at each hash position. We search for a given key key (using a simple lexicographic order on the components) in the corresponding tree. If a term is found, we can return it. If not, we create a new term cell from the `f_code` and the argument terms, and insert that into the tree. During insertion, we compute a number of immutable properties (see next section) that make many operations more efficient. Figure 1 illustrates the basic architecture.

A note on higher-order terms This paper focuses on first-order logic. However, E has recently been extended to higher-order logic [VBCS21, VBS23]. One of the core ideas of this extension was that "you do not pay for features you do not use", in other words, higher-order features should only be visible where strictly necessary. With respect to terms, the two features that directly affect term representation are partial applications and applied variables. For partial applications, we used the fortunate fact that each term cell in E stores the number of arguments of the represented term. In the case of first-order logic that always is the arity of the symbol, and was just added for convenience. But this allows us to represent partial applications by just creating a term with fewer arguments. In other words, if f is a binary function symbol, the term $f@t$ is represented like the first-order term $f(t)$. This corresponds to a flattened *spine* notation [CP03]. Only in the case of applied variables do we resort to an explicit *app*-encoding, using the special variadic function symbol `@_var`, adding the applied variable and the other arguments as proper arguments to the term. Thus, $X@s@t$ is represented as the first-order term $@\_var(X, s, t)$. Finally, λ-terms are supported using a special `f_code` to represent the λ binder, and de-Bruijn-indices to encode bound variables in a locally nameless notation [Cha12]. Each binder abstracts one variable, and de-Bruijn-indices are encoded as constant terms with the `f_code` field overloaded by the index value. De-Bruijn-variables are distinguished by a single-bit term property from normal constants.

2.2 Shared Properties

Having immutable, permanent terms allows us to efficiently pre-compute several term properties, and store them in the term cell. This is particularly true for properties that are normally computed bottom-up. In our case this includes groundness (does the term contain any first-order variables), variable count, function symbol count, and standard weight (computed as two times function symbol count plus variable count). These are used to make many operations more efficient. For example, instantiation does not change ground terms, so when an instance is created we can just return a pointer to the existing term in the term bank for any ground subterm. Standard weight can be used as a cheap pre-test during matching - any instance of a term t always has at least the weight of t, so it is impossible for a heavier term to match a

lighter term.

Temporary shared properties of terms are e.g. variable bindings (computed via unification and matching) and rewrite status (see below).

2.3 Garbage collection

Terms in the term bank are memory-managed via a mark-and-sweep garbage collector. All persistent clause and formula sets are registered with the garbage collector. The system maintains a single *garbage status* bit. All new allocations of term cells are marked with that bit in the current status. If a garbage collection cycle is triggered, the system goes through all registered clauses and formulas, and marks all used term cells by setting a single bit to the complement of the current status. It then goes through the term bank, and frees all cells which still have the current status. Then the global garbage status is flipped. The next collection cycle proceed likewise, only with a different bit value.

In practice, the system performs garbage collection rarely. The collector is triggered during and after clausification, and after unprocessed clauses are culled because the proof state reaches some pre-defined threshold.

3 Cached Rewriting

E uses two different rewrite relations. Both are induced by processed positive unit clauses (also called (potential) *demodulators)*. The first, corresponding to $\Rightarrow_R$ in completion-based system [BDP89], is based only on orientable unit clauses, i.e. clauses in which one side of the single equational literal is already bigger than the other in the term ordering used by the current strategy. Because of the monotonicity of the used orderings, this applies to all instances. Since we only rewrite from larger to smaller terms, we only need to check if the maximal term of such an equation matches to be able to rewrite[1]. The other rewrite relation, corresponding to $\Rightarrow_{R(E)}$, also considers all orientable instances of unit equations. In this case, for unorientable equations we first need to check if either side matches, and then check if the instance generated by the match (possibly after also instantiating unbound variables in the potentially smaller side [Sch22]) is reducing. The second relation is much more expensive to compute, because we need to consider both sides for matching, and in the case of a match, compute a relatively expensive ordering check. Therefore, in most configurations we use the first relation for simplification of the large set of unprocessed clauses, and the second relation only once a clause has been selected for processing, and for back-simplification of the processed clause set. Still, for equational problems, simplification in general and rewriting in particular takes a significant amount of time.

To improve performance at this bottleneck, we have implemented cached rewriting in E, i.e. we store information about rewritability of terms directly at the term node, and reuse it if terms are encountered and need to be simplified more than once. This is similar in spirit to *light normalisation* as implemented in iProver [DK20], but both predates it and is more comprehensive. While iProver caches normal forms for the left hand side of (potential) rewrite rules, E caches rewrite results not at the rule level, but at the term level. Thus, E caches all rewrite steps, while iProver memorizes a shortcut for rewriting with uninstantiated rules.

[1] There are some restrictions on rewriting maximal sides of maximal positive literals in processed clauses, but these are irrelevant to the current discussion.

3.1 Implementation and optimisations

Each term cell carries information about possible rewrites. These consist of two pointers, the `replace` pointer and the `demod` pointer. If the `replace` pointer is not NULL, it points to a term cell representing the term the original has been rewritten to. In that case, the `demod` pointer indicates which clause was used for this rewrite step, thus facilitating proof reconstruction.

If a term with a non-NULL `replace` pointer is encountered during normalisation, the system does not try any demodulators, but simply follows this pointer, pushing the clauses indicated by `demod` pointers onto the modification stack of the clause being simplified.

There are two more optimisations for rewriting built into the term bank. We maintain a monotonically increasing abstract time. In particular, this abstract time always increases when a new clause is added to the set of potential rewrite rules/equations. If a term is found irreducible with respect to the given rewrite relation and the current set of processed unit clauses, we annotate the term with this information (i.e. "Term s is irreducible with respect to all processed orientable unit clauses at time T" or "...with respect to all unit clauses..."). Clauses carry the abstract time they were processed at in their meta-information. If a term is encountered again, and we know that it is irreducible with all clauses at time T, we don't need to try any clauses that have age T or older.

In practice, potential demodulators are stored in indices, trie-like structures where the clauses are stored at the leaves of the tree. We associate each node of this trie with a) the age of the youngest demodulator stored in the subtree rooted there and b) the weight of smallest potentially matching side of demodulators in this subtree. When traversing the tree to find demodulators for a query term, we can ignore all branches only containing clauses that are too old to rewrite the query term, and all clauses whose matching sides are too heavy to match this term.

4 Experimental Results

We ran experiments on all (well-typed, non-arithmetic) first-order problems from TPTP [Sut17], version 8.2.0, for a total of 18102 problems. We recorded a number of statistics for each problem successfully solved, including runtime, number of clauses in the final proof state, number of term nodes assuming unshared terms, number of actual nodes in the shared term DAG representing these, and total number of term nodes in the term bank[2]. Experiments were run on StarExec [SST14], using the StarExec Miami installation. The machines were equipped with 256GB of RAM and Intel Xeon CPUs running at 3.20GHz. We used a 250 second "soft" CPU time limit (i.e. the prover will gracefully terminate after completing the current main loop iteration, providing statistics) and a "hard" limit of 300 seconds . The prover was the first-order version of E 3.0.10 *Shangri-La*, identical to the latest released version of E except for minor bug-fixes and the addition of a number of optional statistics that can be computed and printed after proof search.

We use several different sequential search strategies:

- E's standard automatic mode classifies the problem and then picks parameters that have performed well on similar problems in the past. The major parameters are the clause selection heuristic, determining in which order clauses are picked for processing in the

[2]Our implementation slightly over-counts active DAG notes, because for technical reasons it also counts nodes used by clauses *archived* for proof reconstruction. This is typically a negligible number compared to the overall proof state, but it leads to some visible noise for very small problems. The set of all *term bank nodes* also includes currently unused, i.e. garbage-collectable nodes.

Strategy	Success	Proofs	Saturations	Incomplete
Auto	11453	10268	1185	170
Auto (w/o literal selection)	9406	8734	672	131
Symbol counting 10:1	8935	7825	1110	15
Symbol counting 10:1 (w/o lit.sel.)	7911	7268	643	15

There are 18102 problems in the test set. Incomplete runs are runs where the prover ran out of unprocessed clauses after deleting some (possibly non-redundant) clauses for lack of memory. Proof search for problems not covered by the other columns in each row have terminated unsuccessfully due to timeouts.

Table 1: Performance data for the 4 different strategies

given clause loop, the term ordering, and the literal selection strategy. However, there are many other (mostly binary) parameters that can be set.

- The second strategy is based on the same automatic mode, but explicitly disables negative literal selection, i.e. all maximal literals of a clause are used as inference literals. We chose this option to investigate if the differences in term sharing observed in our 2001 paper [LS01] especially for Horn problems can be confirmed for the current system.
- To minimize the number of variables, we also run an experiment using a single simple but well-performing general purpose strategy. This fixes the term ordering to KBO with weights by inverse symbol frequency rank, precedence by inverse symbol frequency, and constant weight of 1 for constants [Sch22], It uses clause selection using simple symbol counting and clause age in a 10:1 ratio [SM16], and literal selection using `SelectComplex`, a strategy that will always pick a negative inference symbol if available, preferring, in that order, pure variable disequations (i.e. literals of the form $X \not\simeq Y$), the smallest (by symbol count) negative ground literal, and finally the literal with the greatest size difference between the two sides of the literal[3]. We call this *Symbol counting 10:1* or just *SC10:1* below. The term ordering is the one most often used by E in automatic mode (i.e. the one that has performed best over large problem sets in our testing). The literal selection strategy is one of the bests ones that *always* select a negative literal if possible. And finally, the clause selection strategy performas relatively well, follows a scheme that most theorem provers support, and depends only on the signature, not on the conjecture.
- Finally, we ran the same simple strategy, but without enabling negative literal selection.

Table 1 shows the performance data for the different search strategies. For this work, performance is somewhat secondary, but we would like to point out a couple of things. E in automatic mode solves nearly two thirds of all problems. Disabling literal selection reduces this by about 2000 problems, to a bit over one half of all problems. The relatively naive homogeneous strategy with literal selection overall performs similar to auto-mode without literal selection, but does worse for proofs and better for saturations.

Table 2 gives a characterisation of the data we present here. It has four parts, one for each of the four strategies. For each strategy, we present the following measures:

- *Runtime* is the CPU time (in seconds) to completion of the job (either successful or not). Between approximately 100 and 200 runs did not manage to complete in the 300s hard CPU time limit, and thus provided no statistic. These are excluded from the analysis.

[3]E encodes all literals as equations or disequations, using e.g. $p(X) \simeq \$true$ to represent the non-equational literals $p(X)$.

- *Clauses* is the number of clauses in the final proof state, both processed and unprocessed.
- *Term tree nodes* is the number of term cells that would be referenced (directly or indirectly) by the final clauses if E would represent terms as unshared trees.
- *Term DAG nodes* is the number of shared term cells needed to actually represent the above terms (and a small number of terms referenced by archived clauses, see above).
- *All TB nodes* is the number of all term cells stored in the term bank at the time the proof search terminated. In addition to the previous value this includes term nodes that could be garbage collected because they are currently not used by any clause.
- *Sharing factor* is the ratio of *term tree nodes* to *term DAG nodes*.
- *Total rewrites* is the number of successful rewrite steps performed during proof search.
- *Cached rewrites* counts the subset of the previous value that was performed using a cached rewrite link instead of actually finding a fresh demodulator and applying it.
- *Fraction RWs cached* is the ratio of the above, i.e. it gives the fraction of cached rewrite steps relative to all rewrite steps.
- Finally, *TB utilization* is the fraction of all term bank nodes that are referenced by the final proof state, i.e. the fraction of all TB nodes and term DAG nodes.

For each value, we provide the minimum, the first, second (median) and third quartile, and the maximum, as well as the arithmetic mean. For integer values, the average is rounded to the next integer. Note that all values are described independently, i.e. the median value of total rewrites does not necessarily result from the same problem as the median value of the number of cached rewrites, and the median value of the fraction of cached rewrite steps is not the fraction of the median values of cached and all rewrite steps.

We will visualise several of the data distributions in the form of *distribution diagrams*. These diagrams show the values observed in a population of test runs sorted by size - the smallest ones on the left, the biggest ones on the right. Note that because of the great scope of difficulty and run time, in many cases we had to pick a logarithmic y-axis to adequately represent the data.

4.1 Data structures and sharing

Figure 2 shows distribution diagrams for various counts of real or theoretical data structure measures: The number of clauses, number of term tree nodes represented by these clauses, actual term DAG nodes needed to represent them in the shared representation, and nodes actually present in the term bank. The diagram on the left shows data for the normal automatic mode of E (corresponding to Table 2a), the one on the right to automatic mode with negative literal selection disabled (Table 2b).

In both cases we can see that the distribution of clauses and term tree nodes tracks quite well, but that for non-trivial examples, the value for term tree nodes is about two to three orders of magnitude greater than the corresponding number of clauses. This supports the claim about the central role terms play for saturating automated theorem provers.

When we consider shared term cells in the term bank, we can see that both the number of shared cells in the term DAG and of all cells in the term bank again track very closely, with only a relatively small difference between them. They also very roughly track the number of

a) Auto	Min	1st q.	Median	3rd q.	Max	Mean
Runtime	0.0	0.04	1.01	250.92	299.67	91.35
Clauses	0	301	18585	1132492	3094248	521457
Term tree nodes	0	4658	476289	34841300	7393888760	40213831
Term DAG nodes	2	1167	16399	659293	8665762	471810
All TB nodes	2	1554	20948	772362	11615296	605027
Sharing factor	0	3	15	53	13080680	2285
Total rewrites	0	53	6866	1323619	2350108946	3075329
Cached rewrites	0	35	5754	1201231	2348276767	2822675
Fraction RWs cached	0.0	0.579476	0.862697	0.972625	1.0	0.715044
TB utilization	0.000015	0.747881	0.890252	0.965839	1.0	0.832692
b) Auto w/o lit.sel.	Min	1st q.	Median	3rd q.	Max	Mean
Runtime	0.0	0.06	30.94	251.35	290.25	122.05
Clauses	0	706	511220	1301428	3037819	692984
Term tree nodes	0	12268	17398621	47747811	7437234556	49428700
Term DAG nodes	2	1874	35424	365138	7200828	330910
All TB nodes	2	2254	40074	393308	10918291	416376
Sharing factor	0	7	42	243	13080680	2880
Total rewrites	0	68	16919	731660	2350108946	2223361
Cached rewrites	0	43	15645	700882	2348276767	1999820
Fraction RWs cached	0.0	0.666667	0.92072	0.989583	1.0	0.750739
TB utilization	0.000015	0.82482	0.942162	0.994384	1.0	0.877701
c) SC 10:1	Min	1st q.	Median	3rd q.	Max	Mean
Runtime	0.0	0.05	249.95	251.13	283.02	130.72
Clauses	0	757	538654	1464193	2929585	777191
Term tree nodes	0	12515	15874006	42004063	9297406274	47137260
Term DAG nodes	2	2179	259782	2216163	9123313	1163082
All TB nodes	2	2680	324531	2365703	12303650	1286958
Sharing factor	0	5	11	26	617798	1071
Total rewrites	0	208	42766	1901498	285140568	4513105
Cached rewrites	0	111	36574	1718916	277525335	4148014
Fraction RWs cached	0.0	0.600037	0.867287	0.97038	1.0	0.723709
TB utilization	0.002204	0.820367	0.940815	0.995384	1.0	0.877509
d) SC 10:1 w/o lit.sel	Min	1st q.	Median	3rd q.	Max	Mean
Runtime	0.0	0.11	250.74	251.46	296.54	144.05
Clauses	0	2348	998417	1492407	2986997	875239
Term tree nodes	0	56315	32810594	61388806	9447774986	64101714
Term DAG nodes	2	2936	86396	703796	9123314	549414
All TB nodes	2	3558	93611	729518	11051673	639408
Sharing factor	0	11	57	241	3032020	1800
Total rewrites	0	276	106863	1163220	285140568	3084820
Cached rewrites	0	193	97326	1085361	276724411	2811925
Fraction RWs cached	0.0	0.724032	0.942955	0.988776	1.0	0.781862
TB utilization	0.002204	0.883698	0.97377	0.996761	1.0	0.905721

Table 2: Overview of result data

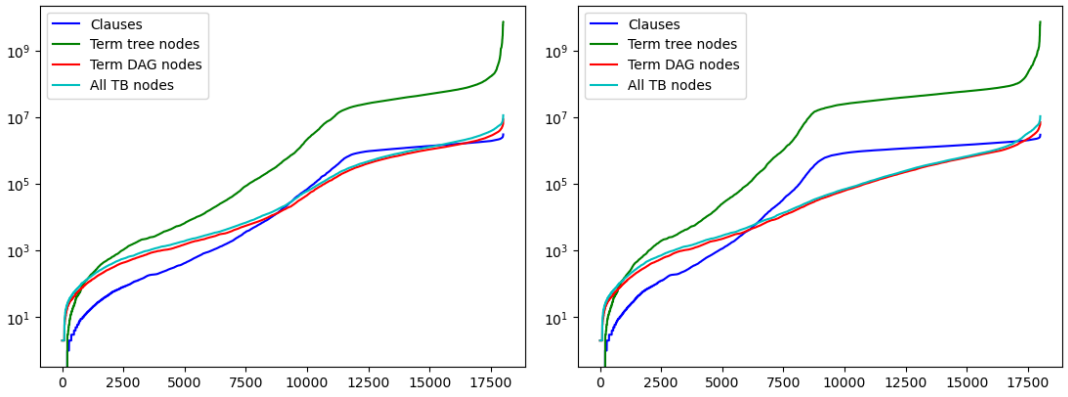

Figure 2: Distribution of clause number and different term nodes counts for auto-mode (left) and auto-mode without literal selection (right)

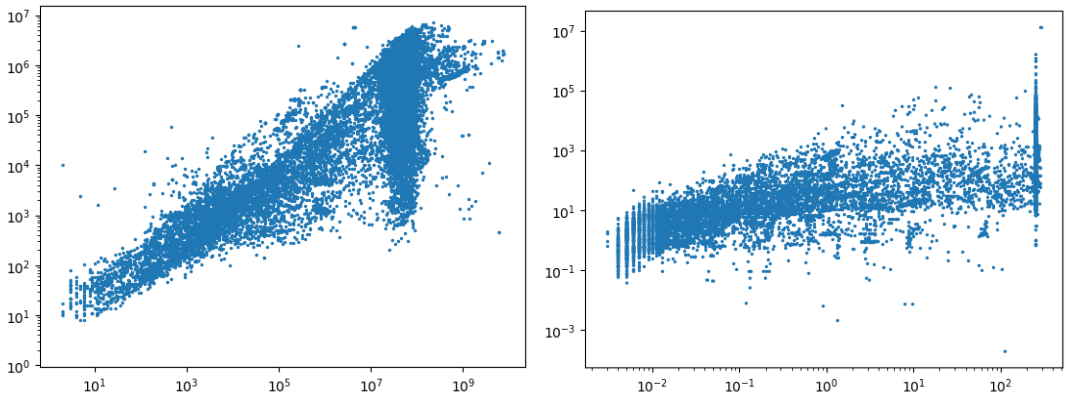

Figure 3: Scatter plot of term DAG nodes over term tree nodes for automatic mode (left) and of sharing factor over runtime (right). Notice that both axes are logarithmic for both diagrams.

clauses, but with a lot more variation. However, especially for harder problems (i.e. problems for which the prover needs a longer time to complete) with greater number of both clauses and term cells, we can see that shared term counts often are lower than clause counts.

This great saving in the number of term cells is confirmed if we consider the actual values of shared term cells relative to unshared tree cells. Figure 3 (left) visualises this data. Each dot corresponds to a single problem (run in automatic mode), with the x-coordinate determined by the number of (theoretical) term nodes in an unshared tree representation, and the y-coordinate representing the number of nodes in the shared representation. This diagram style allows us to see the wide spread of relative values, but it also confirms that the about 2.5 orders of magnitude for non-trivial problems is typical.

The right diagram in Figure 3 visualises and compares the distribution of the *sharing factor* (i.e. the ratio of term tree nodes to term DAG nodes) for all 4 different search strategies. This factor tells us how many unshared nodes a shared node typically represents, or in other words, the relative memory increase an unshared term representation would cause. For non-

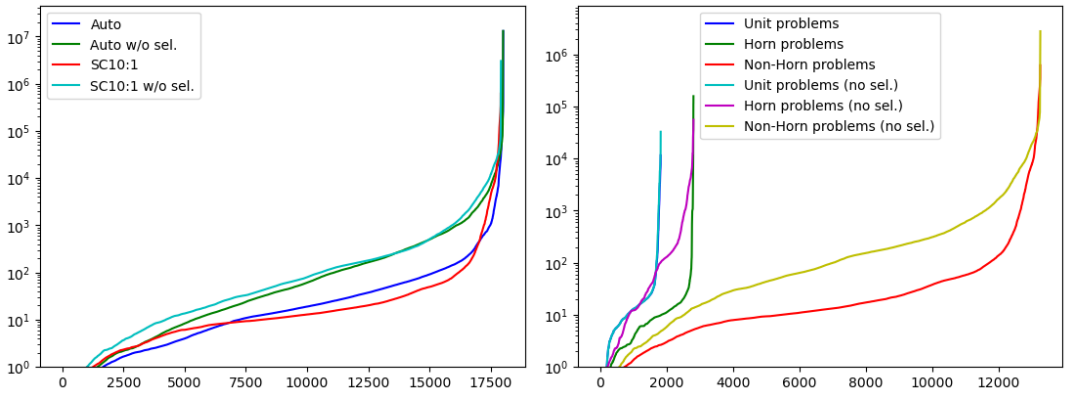

Figure 4: Distribution of the sharing factor for all four search strategies (left) and for Unit, Horn and non-Horn problems for the symbol counting strategy with and without literal selection (right)

trivial and non-extreme problems, the sharing factors vary between ≈10 and ≈100, but for harder problems, it often reaches the thousands, and in the extreme case several millions. There also is a significant number where the recorded sharing factor is well below 1 even for non-trivial problems. We have investigated some of these cases, and they stem from examples where the prover produces a very small final clause set (usually a saturation or incomplete saturation), but with a non-trivial derivation. The most extreme example comes from the TPTP problem `COL125+1.p`. The problem has status *CounterSatisfiable* (i.e. the resulting clause set is satisfiable) and prover eventually derives the final single-literal clause $X1 \simeq X2$, which subsumes all other clauses, leading to a final proof state with just 2 term cells. However, the derivation of that final clause is highly non-trivial, and there 1684 archived clauses that are kept to enable proof reconstruction. As noted above, the term cells referenced by clauses in this set are counted against the shared term cell counts, resulting, in this case, to a significant overcount.

Another interesting aspect becomes apparent if we compare the distributions for the different strategies. The two non-literal-selecting strategies behave very similar, as do the two literal-selecting ones. In general, sharing is a lot higher for the non-selecting strategies. This tracks with our earlier results [LS01] and seems to indicate that negative literal selection not only finds more proofs faster, but also that it results in less redundancy in the generated terms. We can also see the effects in the numerical data in Table 2. We have visualised the distributions for individual problem classes in Figure 4. As expected, literal selection has no effect (except for random noise) on unit problems (the blue data points are nearly perfectly covered by the cyan line). For both Horn and non-Horn problems we can see that literal selection drastically lowers the sharing factor, but even more so in the Horn case.

4.2 Garbage collection

Figure 5 gives us some insight into the amount of collectable (i.e. not currently referenced) term cells in the term bank. On the left diagram, we can see that for the vast majority of problems, the two values - utilized and all term cells - lie very close together, placing the data point on or just below the diagonal. There are, however, a few clusters of problems where the number

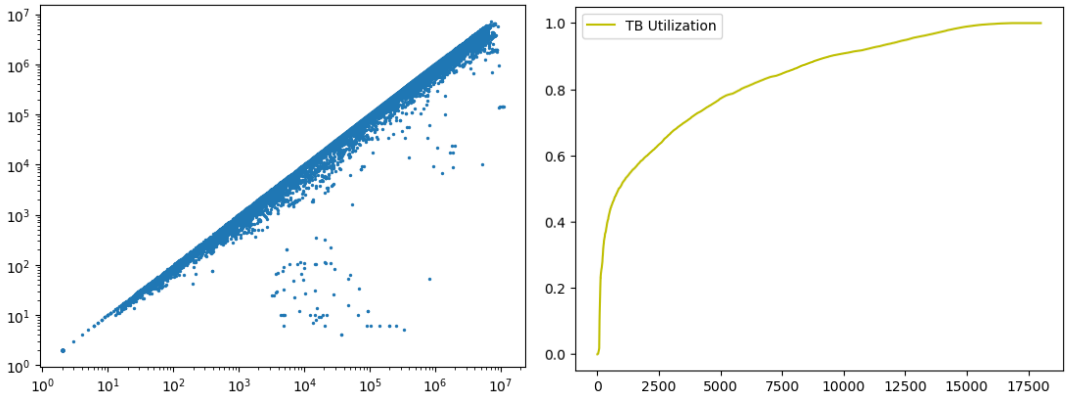

Figure 5: Referenced term bank nodes over all term bank nodes (left) and distribution of the utilization fraction (right) for automatic mode

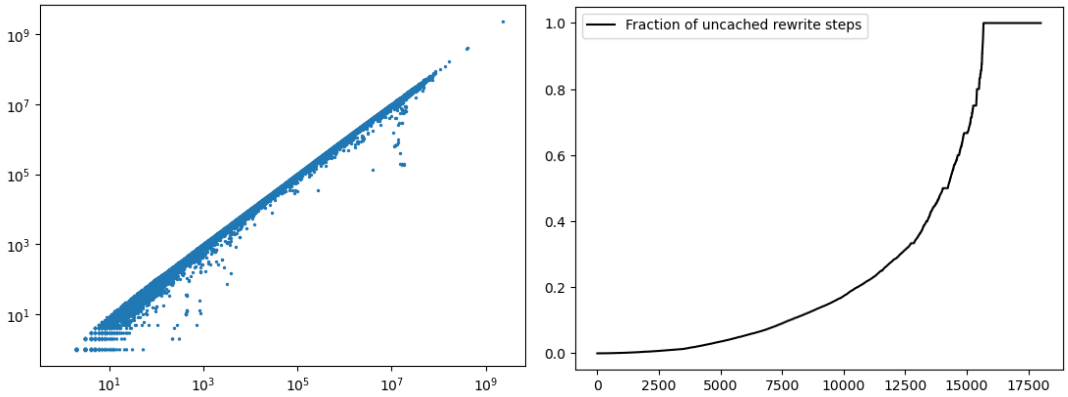

Figure 6: Cached rewrites over all rewrites (left) and distribution of the fraction of uncached rewrites (right) for automatic mode

of used nodes is significantly lower than the number of all stored nodes in the term bank. In theorem proving we sometimes observe that a few critical rewrite rules, once derived, can lead to a big collapse in the proof state, as very many clauses can suddenly be simplified. Similarly, sometimes a key clause can be derived that subsumes a large number of other clauses. Either of these would explain the outlying clusters.

On the right hand side, we see the distribution of the term bank utilization over all problems. Only very few problems show a utilization of less than 50%, and for most problems this factor is over 80%. Table 2 confirms this, with the median term bank utilization between 89% and 97% (depending on the search strategy). Overall, we conclude that our decision to only trigger garbage collection in specific situations is adequate, and that most term nodes that are created are in use over a long time.

4.3 Cached rewriting

Finally, Figure 6 visualises some of the data on cached rewriting. On the left, we can see a scatter plot showing the number of cached rewrites over the number of all rewrites. As we can see, the "main sequence" follows the diagonal, with the spread of values becoming smaller as the number of rewrite steps increases. In other words, the more rewrite steps there are, the higher the percentage of those that are cached, There are, however, a number of outliers.

The diagram on the right shows the distribution of the fraction of uncached rewrite steps. The median of this distribution (for automatic mode) is 13.7%, or about 1 in 8 rewrite steps. However, as seen above, most of the more difficult problems have a much lower fraction of uncached steps.

5 Lessons Learned

As E was originally built with the dynamic term banks in mind, we allowed for multiple term banks to be in use (because e.g. some terms need to be preserved while others are rewritten). We also allowed for multiple instances of the same term, only distinguished by some single-bit properties. Both of these features are no longer used with the new immutable terms and cached rewriting driven from the clause level. By designing a prover around a single term bank distinguishing terms by structure only, quite a bit of simplification would be possible. In particular, we could always use pointer identity as syntactic identity for shared terms, without careful thought about where the terms come from.

Also, strict commitment to have all non-transient terms shared would make most support for unshared terms, in particular for parsing them, unnecessary. A trivial implementation improvement would be to include the term bank pointer into the term data structure (for all shared terms). It is needed nearly everywhere terms are processed, and the pointer could thus be made easily available, and serve as a marker to distinguish shared terms from temporary unshared ones when needed.

A number of features of E's shared terms were either never used, or have long since fallen into disuse. This included the ability to print and parse terms in an abbreviated fashion (using *node id*s to represent shared subterms), and the ability to parse and print Prolog-style lists. Also, E now supports the old LOP-format, two different TPTP syntaxes for first-order logic, the later also in a typed variant, and, after extension to higher-order logic [VBS23] the (largely independent) TPTP syntax for monomorphic higher order syntax [SB10]. In a re-implementation, it would probably be better to concentrate on the modern TPTP syntax [SSCB12, SB10], and to keep the parsers for first-order and higher-order logic largely separate.

Indexing with weight and age constraints could be applied more consequently, and would profit from the lazy approach to update constraints described previously [Sch24].

We consider it an open question if (equational) literals should be represented as shared terms at the clause level. This would have some advantages, but the greater freedom of adding useful information at the literal level also has its value. Also, equations are usually unordered term pairs, so they would still need special handling in many situations.

Managing term memory with garbage collection has been a particularly productive idea. It frees developers from manually tracking references, and allows them to simply construct and discard terms as is convenient. Indeed, the impact of garbage collection on term cells was so big that we replaced E's native and distinct formula data type with term-encoded formulas (where logical operators and quantifiers are just special interpreted function symbols). This made the later move to logics with first class Booleans [SCV19, VBCS21] like TF0 and FOOL [KKRV16],

where formulas and terms become one structure anyways, much easier.

6 Conclusion

The choice to go with a shared term data structures has paid off for E in multiple ways. As demonstrated in this paper, for hard problems we achieve massive savings in the number of term cells, typically to a degree that the number of term cells is of the same order of magnitude as the number of clauses, and hence no longer the limiting factor.

High levels of term sharing can be observed over nearly all problem types and all non-trivial problems, but it seems to go up with the number of terms and, though with a larger spread, with runtime. In general, high levels of sharing seem to indicate a lot of redundancy in the proof state - this is particularly obvious if we compare the (usually) stronger calculus variants with literal selection to the ones without. There may be a way to utilise this fact to help control proof search in the future, but so far this remains a vague idea.

Cached rewriting has shown good potential, reducing the number of expensive new rewrites by orders of magnitude for hard problems. It would be interesting to analyse how often size and age constraints have cut short the search for demodulators early, but that is beyond the scope of this paper.

A substantial amount of experience was accumulated with shared terms and cached rewriting in E. We hope that this paper helps future implementation to avoid some of the pitfalls along the way and build on our experience.

References

[BDP89] L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion Without Failure. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 1–30. Academic Press, 1989.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[Cha12] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, 2012.

[Chr93] J. Christian. Flatterms, Discrimination Nets and Fast Term Rewriting. *Journal of Automated Reasoning*, 10(1):95–113, 1993.

[CP03] Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

[DK20] André Duarte and Konstantin Korovin. Implementing superposition in iProver (system description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Proc. of the 10th IJCAR, Paris (Part II)*, volume 12167 of *LNAI*, pages 158–166. Springer, 2020.

[DKS97] J. Denzinger, M. Kronenburg, and S. Schulz. DISCOUNT: A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, 18(2):189–198, 1997. Special Issue on the CADE 13 ATP System Competition.

[KKRV16] Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. The Vampire and the FOOL. In Jeremy Avigad and Adam Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, USA*, pages 37–48. ACM, 2016.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Proc. of the 25th CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

[LH02] B. Löchner and Th. Hillenbrand. A Phytography of Waldmeister. *Journal of AI Communications*, 15(2/3):127–133, 2002.

[LS01] B. Löchner and S. Schulz. An Evaluation of Shared Rewriting. In H. de Nivelle and S. Schulz, editors, *Proc. of the 2nd International Workshop on the Implementation of Logics*, MPI Preprint, pages 33–48, Saarbrücken, 2001. Max-Planck-Institut für Informatik.

[McC10] William W. McCune. Prover9 and Mace4. http://www.cs.unm.edu/~mccune/prover9/, 2005–2010. (acccessed 2016-03-29).

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[RW69] G. Robinson and L. Wos. Paramodulation and Theorem Proving in First-Order Theories with Equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, 1969.

[SB10] Geoff Sutcliffe and Christoph Benzmüller. Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure. *Journal of Formalized Reasoning*, 3(1):1–27, 2010.

[Sch02] Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[Sch22] Stephan Schulz. Empirical properties of term orderings for superposition. In Boris Konev, Claudia Schon, and Alexander Steen, editors, *Proc. of the 8th PAAR, Haifa, Israel*, number 3201 in CEUR Workshop Proceedings, 2022.

[Sch24] Stephan Schulz. Lazy and eager patterns in high-performance automated theorem proving. In Laura Kovács and Michael Rawson, editors, *Proceedings of the 7th and 8th Vampire Workshop*, volume 99 of *EPiC Series in Computing*, pages 7–12. EasyChair, 2024.

[SCV19] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *Proc. of the 27th CADE, Natal, Brasil*, number 11716 in LNAI, pages 495–507. Springer, 2019.

[SM16] Stephan Schulz and Martin Möhrmann. Performance of clause selection heuristics for saturation-based theorem proving. In Nicola Olivetti and Ashish Tiwari, editors, *Proc. of the 8th IJCAR, Coimbra*, volume 9706 of *LNAI*, pages 330–345. Springer, 2016.

[Sma21] Nick Smallbone. Twee: An Equational Theorem Prover. In André Platzer and Geoff Sutcliffe, editors, *Proc. of the 28th CADE, Pittsburgh*, volume 12699 of *LNAI*, pages 602–613. Springer, 2021.

[SSCB12] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Peter Baumgartner. The TPTP Typed First-order Form with Arithmetic. In Nikolaj Bjørner and Andrei Voronkov, editors, *Proc. of the 18th LPAR, Merida*, volume 7180 of *LNAI*, pages 406–419. Springer, 2012.

[SST14] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A Cross-Community Infrastructure for Logic Solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Proc. of the 7th IJCAR, Vienna*, volume 8562 of *LNCS*, pages 367–373. Springer, 2014.

[ST85] D.D. Sleator and R.E. Tarjan. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32(3):652–686, 1985.

[Sut17] Geoff Sutcliffe. The TPTP problem library and associated infrastructure - from CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[VBCS21] Petar Vukmirović, Jasmin Christian Blanchette, Simon Cruanes, and Stephan Schulz. Extending a Brainiac Prover to Lambda-free Higher-Order Logic. *International Journal on Software Tools for Technology Transfer*, August 2021.

[VBS23] Petar Vukmirović, Jasmin Christian Blanchette, and Stephan Schulz. Extending a high-performance prover to higher-order logic. In Natasha Sharygina and Sriram Sankaranarayanan, editors, *Proc. 29th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'23), Paris, France*, number 13994(2) in LNCS, pages 111–132. Springer, 2023.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS Version 3.5. In Renate Schmidt, editor, *Proc. of the 22nd CADE, Montreal, Canada*, volume 5663 of *LNAI*, pages 140–145. Springer, 2009.