EPiC Series in Computing

Volume 100, 2024, Pages 106–115

Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning

Waste Reduction: Experiments in Sharing Clauses between Runs of a Portfolio of Strategies (Experimental Paper)

Tanel Tammet

Applied Artificial Intelligence Group,
Tallinn University of Technology, Tallinn, Estonia
tanel.tammet@taltech.ee

Abstract

Most high-end automated theorem provers for first order logic (FOL) split available time between short runs of a large portfolio of search strategies. These runs are typically independent and can be parallelized to exploit all the available processor cores. We explore several avenues of re-using clauses generated by earlier runs and present experimental results of their usefulness or lack thereof.

1 Introduction

Since 1996, the yearly CASC competition [17] of theorem provers has been the main event for comparing their performance on highly varied types of problems, selected from the TPTP collection [24]. During the early days of CASC, an important focus of the research in the area was on core search algorithms, particularly the resolution method, and specialized datastructures, particularly indexes for quick retrieval of candidates for various operations. However, it soon started to become clear that implementing a large number of different specialized strategies and search restrictions along with a well-tuned selection mechanism for these strategies is more important than the technological advancements in core algorithms.

Since the heuristics for selecting a suitable strategy for a given problem turned out to be poor – and have arguably remained so – the Gandalf prover [21] for FOL, implemented by the author, introduced time slicing. Instead of using just one search strategy for one problem, Gandalf selected a set of different search strategies, allocated time to these and finally ran the strategies one after another. Time slicing made Gandalf the winner of the FOL category of CASC in 1997 and 1998. Since then, time slicing with a heuristically selected large portfolio of strategies has been a basic principle of all high-performance FOL provers. The same principle is is also used in a number of different search areas, like propositional theorem proving.

When running one strategy after another for a same problem, then clauses derived during one run could potentially be kept and used for next runs as well. However, this poses both conceptual and technological problems.

To our knowledge, high-performance provers participating in the CASC competition do not employ clause sharing between different runs, except for – possibly – specialized propositional methods used in the state-of-the art prover Vampire [10], as reported and analyzed in [14] and

[12]. Thus, each run is essentially a fresh start and all the derived clauses are discarded. Interestingly, the early Gandalf prover was an exception in this respect: it did keep a heuristically selected subset of derived clauses for use in later runs.

The earliest implementation of re-use of clauses known to the author is the DISCOUNT system [1], [5], based on the TEAMWORK concept [4] of Denzinger for distributing deduction systems on several processors. A later application of re-using clauses is the use of Prover9 [11] in loop theory. The paper [9] describes the utility `p9loop` which runs Prover9 iteratively on a list of different term orderings using a *hint mechanism* of Prover9. New iterations include all of the previous hint matchers as additional input assumptions. The paper notes that this approach has been successful, and sometimes found proofs after 50 iterations.

Several machine learning methods for proof search also re-use clauses. In [8] lemma selection from the derived clauses is used between related problems, while in [3] a similar mechanism, called *leapfrogging*, is used for E/ENIGMA between repeated runs on the same problem. The latter paper describes the use of a trained graph-based predictor to select a promising subset of the clauses processed during a previous run.

Regarding Gandalf, the current author concluded, citing the paper describing the main principles of Gandalf [22]: "However, for the CASC-14 competition examples this cooperation was very rarely of any use."

This early claim was not strongly grounded on careful experiments, and may not necessarily hold for current FOL provers. Thus the purpose of this paper: to explore several avenues of re-using clauses generated by earlier runs and present experimental results of their usefulness or lack thereof.

The version of GKC used for experiments along with the experimental data can be found from https://github.com/tammet/gkc/tree/master/Share.

2 GKC: The Underlying Implementation

We conduct experiments on a modified version of the conventional high-performance resolution-based FOL prover GKC [18] implemented by the author and available at https://github.com/tammet/gkc. The prover was built to serve as an underlying system for the commonsense reasoner GK, which implements numeric confidences [20] and default logic [19]. The GK system is a critical component of the pipeline [23] for commonsense reasoning using natural language.

The base prover GKC itself is a capable prover optimized for search in large knowledge bases. It has performed relatively well in the recent CASC competitions of provers [17], achieving the fifth position among the eleven competitors in the First-order Theorems category of the CASC 29 competition held in 2023, and a similar position in all the CASC competitions it has participated in, starting from 2019. We note that the equational reasoning components of GKC are relatively weak and hence it is typically one of the weakest competitors in the Pure Unit Equality category.

GKC is implemented in C on top of the data structures and functionality of the shared memory database WhiteDB, see https://github.com/priitj/whitedb. GKC is available for both Linux and Windows under GNU AGPLv3.

The derivation rules currently implemented in GKC are highly common in FOL provers. For the standard terminology, see [2].

1. Binary resolution with optionally the set of support strategy, negative or positive ordered resolution, unit restriction and a wealth of other restriction strategies.

2. Factorization.

3. Paramodulation with the Knuth-Bendix ordering.

4. Demodulation (equational rewriting).

5. SInE preprocessing and selection, see [7].

In particular, we note that no purely propositional methods like AVATAR [13] have been implemented so far. We do not perform subsumption of given clauses with non-unit clauses in the knowledge base: during our experiments the time spent for this did not give sufficient gains for the efficiency of proof search.

The overall iteration algorithm of GKC is based on the common *given-clause algorithm* where newly derived clauses are pushed into the *passive list* and then selected (based on the combination of creation order, clause weight and other parameters) as a *given clause* into an *active list*. The derivation rules are applied only to the given clause and active clauses. In the following we explain how we perform clause simplification with all the derived clauses present in the active list. We will also explain the use of different clause selection queues used by GKC.

One of the innovations in GKC is the pervasive use of hash indexes instead of tree indexes. In contrast, all state-of-the-art provers implementing resolution rely on tree indexes of various kinds. Research into suitable tree indexes has been an important subfield of automated reasoning.

Our experiments demonstrate that hash indexes are a viable alternative and possibly a superior choice in the context of large knowledge bases. The latter are expected to consist mostly of ground clauses representing known "facts" and a significant percentage of derived literals are ground as well.

Another innovation of GKC is the selection of a given clause by using several queues in order to spread the selection relatively uniformly over different important categories of derived clauses. This technique has been later explored by Martin Suda in [6], who also implemented it in Vampire [10]. A similar technique has been implemented earlier in the E prover, see [15] and [16].

The queues are organized in two layers. As a first layer we use the common ratio-based algorithm [6] of alternating between selecting N clauses from a weight-ordered queue and one clause from the FIFO queue with the derivation order. This *pick-given ratio* N is set to 4 by default.

As a second layer we use four separate queues based on the derivation history of a clause. Each queue in the second layer contains the two sub-queues of the first layer.

The formulas in the TPTP collection [24] are normally annotated as either being axioms or conjectures/goals to be proved or assumptions/hypothesis posed and relevant for the goal. Hence we split all the input and derived clauses into four classes based on their history according to the annotations:

1. Clauses having both the goal and assumption in the derivation history.

2. Clauses having some goal clauses in the derivation history.

3. Clauses having some assumption clauses in the derivation history.

4. Clauses having only axioms in the derivation history.

These four queues are disjoint and if conditions are not mutually exclusive, the higher (earlier) one has priority.

The different runs of a portfolio strategy are implemented using forks. The main procedure first parses the problem, creates necessary data structures and determines the portfolio of strategies. Then it spawns N parallel processes, giving a selection of strategies for each process.

3 Shared Memory

The sharing of clauses between different runs is implemented using shared memory. Achieving efficiency of the implementation is simplified by the fact that core datastructures of GKC use the shared memory database WhiteDB. WhiteDB is a lightweight NoSQL database library written jointly by the author of this paper and Priit Järv as a separate project. It is available under the GPL licence and can be compiled from C source or installed as a Debian package.

Each term or clause is represented as a WhiteDB database record containing meta-information followed by term elements encoded as integers. WhiteDB database records – and hence also the main data structures in GKC – are tuples of N elements, each element encoded as an integer in the WhiteDB-s data encoding scheme. Since the WhiteDB data structures can be kept in the shared memory where absolute pointers do not work (processes map memory areas to different address spaces), conventional pointers are not used in the main data structures: this role is given to integers indicating offset to the current memory area, thus enabling the data in memory to be independent of its exact location.

The shared memory database makes it possible to start solving a new problem in the context of a given knowledge base very quickly: parsing and preprocessing the large knowledge base can be performed before the GKC is called to do proof search: the preprocessed database can be assumed to be already present in the memory.

During the experiments described in the current paper, GKC does not only use the shared memory for avoiding duplicate parsing and indexing, but also writes selected clauses to shared memory for subsequent use by later runs.

4 Sharing and Using Clauses between Runs

There are several possible avenues of re-using clauses derived during the many runs of a portfolio of strategies for a single problem. For example, a common pattern of using a portfolio is to run N strategies with a time limit t, and then run the same strategies again with an increased time limit $t * c$, etc.

One could envision storing a whole search state for one run, to be later continued with an increased time limit. This would, however, be complex to implement, require a large amount of memory and possibly incur significant time penalties. Another option would be storing clauses strictly for simplification, like demodulation (equational simplification) or subsumption resolution (replacing a clause with a subsuming instance resulting from a resolution step).

A simple form of subsumption resolution ("cutoff with a unit" in the following) is cutting a literal L off a clause $L \vee \Gamma$ with a unit clause $\neg L'$, in case L' subsumes L. GKC employs an even simpler, essentially propositional form: the atom L' in the unit clause $\neg L'$ must be equal / same as L which is cut off the clause.

Similarly, GKC employs an essentially propositional form of subsumption resolution with two-literal clauses ("cutoff with a two-literal clause" in the following) : a literal L is cut off a

clause $L \vee R \vee \Gamma$ with a two-literal clause $\neg L' \vee R'$, in case L' is equal / same as L and R' is equal / same as R.

GKC checks and performs all possible cutoffs with a unit for all the derived clauses after they have passed the initial retention tests like unit subsumption. For this it uses all the kept units, searching for equal literals using hashes. The motivation is to improve the chances of a thus simplified and lighter clause to be selected as a given clause earlier. Additionally, GKC performs all possible cutoffs with a unit for each given clause.

The cutoffs with a two-literal clause are performed only for given clauses, and not for all freshly derived clauses, which has thus a smaller effect on proof search.

The experimental inter-run sharing strategy for unit and two literal clauses means copying all the derived unit and two-literal clauses to a WhiteDb shared memory database, along with corresponding additions to hash-based data structures, after a run has been finished. All the accumulated unit and two-literal clauses will henceforth participate in the simplification operations described above. An important omission in copying is the history of the clause, i.e. parents: these are not copied.

A fair question to ask is whether using clauses from previous runs for simplification may lose completeness, i.e. proofs which could otherwise be found, may become impossible to find. Differently from a normal situation, in our algorithm the simplifying unit clauses do not enter search space. First, we note that in our system we do not use the clauses from previous runs for equational simplification, i.e. demodulation, which could lead to issues when different runs use different ordering strategies. Second, a significant percentage of our runs use strategies which do not preserve completeness anyway.

The completeness of simplification by shared clauses can be proved by reducing it to the classic question of completeness of subsumption, Ordinary search strategies, with the exception of very specific ordering strategies, are known to be complete under subsumption. Suppose a shared clause $\neg L$ is used to cut the literal L from a clause $L \vee \Gamma$, leaving just the subsuming Γ. Instead of performing the cut, we could have entered the clause Γ into search space at the same moment, either replacing $L \vee \Gamma$ if the latter is a currently selected active clause, or otherwise later, when $L \vee \Gamma$ would have been selected as an active clause. We could simulate Γ entering the search space by a specialized clause selection strategy, where Γ is among the input clauses, but is not selected as an active clause until the cut is performed. This argumentation seemingly does not always hold for the set of support strategy, where a subset S of all input clauses I is never selected, but is considered to be active already from the beginning. However, in this case we could consider Γ to be an input clause in the initially inactive set $I - S$, and selected later during the proof search.

In order to enable parallel runs while sharing is active, we employ the built-in locking mechanisms of the WhiteDB database: the global write lock is asked for / set once before the whole copying procedure starts, to be released upon finishing the copying. Any time we search for unit cutoffs from the shared area, we ask for / set a read lock, to be released after the search. In our experiments, copying the unit clauses is fast and thus the global write lock does not appear to significantly diminish performance. Copying two-literal clauses is slower – there are typically more of these – and may start to affect the performance, in particularly, due to locking, should we implement it for two-literal clauses.

Due to technical complexities we have not implemented locking of two-literal and longer clauses, thus our experiments with these are limited to sequential runs of the portfolio strategy.

Finally, we also experiment with sharing full clauses, to be used as a part of a set of input clauses for later runs. Incidentally, this was implemented in the old Gandalf prover [21]. Further details are given in the next section.

The described scheme has an important omission regarding the parents of the copied and shared clauses, which fortunately does not hinder our experiments. Thus, when we output a proof for a problem, we cannot output the whole proof: the parents of the shared clauses are not known. Efficiently storing the parents of the shared clauses is a nontrivial technical problem we have not taken up yet.

5 Experiments

As a test set, we use the 500 problems of the first-order division FOF of the latest CASC competition from 2023. Our prover GKC participated in the competition and finished fourth among the different prover codebases. Vampire and E were represented by two different version each, of which we list the best ones only. As a comparison, Vampire proved 451 (90%) of the problems, E 393 (78%), iProver 354 (70%), GKC 310 (62%), Drodi 301 (60%), cvc5 297 (59%), Zipperposition 269 (53%). The remaining four provers not listed above were significantly weaker.

5.1 Base Performance: no Sharing

Our experiments were run on a Lenovo laptop with the Intel i7-10875H octa-core CPU and 32 gigabytes of memory, under Ubuntu 20.04.6 LTS. This is a faster machine than the one used during CASC. In order to make our results more comparable with the CASC, we experiment under stricter time limits than CASC, which gave 120 seconds of wall-clock time for each prover. It is worth noting that increasing the time limit leads to quickly diminishing gains: of all the 310 problems solved in CASC, GKC proved only 14 with a running time over 60 seconds. During CASC, GKC used eight parallel processes for each problem.

First we establish the base performance on our test computer without sharing clauses between runs. With a time limit of 240 seconds and no parallel processes, GKC finds 294 proofs. With a time limit of 60 seconds and four parallel processes, GKC, unsuprisingly, finds the same number of proofs, although not exactly the same ones: both versions found two proofs not found by another. This is to be expected: coupled with various timing checkpoints employed by GKC, small changes in performance lead to semi-random perturbations of proof search.

5.2 Sharing Unit Clauses

Next we test sharing of unit clauses only. Recall that for each problem the prover performs a large number of short independent search attempts we call runs. After each run, all the unit clauses generated during the run are added – by copying – along with corresponding additions to hash indexes, to a shared memory area, thus accumulating during the whole search process. Each run uses the unit clauses for simplification: both the ones generated during the run and the ones accumulated in the shared memory.

With a time limit of 60 seconds and four parallel processes, sharing unit clauses leads to 314 proofs found: 20 more than the non-sharing parallel search. Importantly, the gain is not monotone: although sharing produces new proofs, it also fails to find some proofs found without sharing. Concretely, unit sharing gains 36 proofs while it loses 16 proofs.

When looking at the successes and failures for particular classes of problems, three problem classes stand out. Geometry problems (GEO in TPTP) gained eight proofs, while losing one. “Software verification” (SWV in TPTP) gained four proofs, while losing none. On the other hand, “Software verification continued” (SWW in TPTP) lost five proofs, while gaining one.

The reasons for losing some of the proofs may be twofold: either copying and simplification spend too much time or the simplification sometimes leads the proof search to less advantageous paths. Knowing the approximate time spent on copying and simplification is relatively small, we suspect the perturbations to be the main reason.

The following table gives an overview of several additional experiments: using a different number of parallel processes and imposing limits on the number of shared unit clauses. Each line compares proof searches without sharing to searches with sharing, with the same wall clock limit and the number of parallel processes for non-sharing and sharing runs. The first column indicates total wall clock time, the second column indicates the number of parallel processes and the third gives a number of problems proved without sharing. The "limits" column contains either "-" for no limits to shared units, "processed" for using only processed (aka given or active) unit clauses for sharing, or a maximum number of positive and negative units allowed to be shared, with separate limits for each. In the latter case the selection of actually shared units is essentially random. The final three columns indicate the effect of sharing: proofs gained, proofs lost and the difference between the latter two. Observe that for all but the last row, the wall clock time multiplied by the number of processes is always 240. The last row is included to show the effects of increasing this number twofold, to 480.

seconds	parallel	no sharing	limits	sharing: gained	sharing: lost	total gain
240	1	294	-	+36	-13	23
60	4	294	-	+35	-15	20
60	4	294	processed	+36	-15	21
60	4	294	10000	+36	-14	22
60	4	294	1000	+36	-16	20
60	4	294	100	+34	-18	16
30	8	289	-	+32	-19	13
60	8	307	1000	+31	-17	14

The table shows that as the amount of parallelism increases, the positive effects of unit sharing slowly diminish. Interestingly, relatively harsh limits on the number of shared clauses do not give a seriously negative effect, and a relatively high limit of 10.000 per positive and negative units shared actually increases the performance slightly. This seems to indicate that the main gain from unit sharing may rise from keeping units generated by strategies which generate relatively few unit clauses.

5.3 Sharing Unit and Two-Literal Clauses

After testing unit sharing, we consider sharing both unit and two-literal clauses. Differently from unit clauses, GKC does not perform immediate simplification of freshly generated clauses with stored two-literal clauses: the latter are used for simplifying given clauses only. Importanly, due to technical complexities, we have not implemented sharing of two-literal clauses in the context of parallel processes: thus the experiment is run with a time limit of 240 seconds and a single process, i.e. strictly sequential runs. This produces 306 proofs: more than the base performance with no sharing (294 proofs) and less than sharing unit clauses only (314 proofs). When compared to sharing only unit clauses, sharing both units and two-literal clauses gains 2 proofs while losing 10 proofs. Importantly, while the time cost of sharing unit clauses appeared to be small, the time cost of sharing two-literal clauses was noticeably bigger. The reasons for performance worse than just unit sharing are likely stemming from both the time cost and the semi-random perturbations of proof search.

5.4 Sharing Full Clauses

Finally, we experimented with additionally sharing full clauses between runs: we share both unit clauses and two-literal clauses for simplification as before, plus sharing full derived clauses for inclusion in the clause set. Again, we have not implemented sharing of full clauses for parallel runs, thus all the experiments were run with a 240 second time limit and no parallelism.

The way full clauses are selected for sharing and then used during search differ substantially from sharing unit and two-literal clauses. While the latter were used strictly for simplification, shared full clauses were entered as a part of the clause set of the problem. In other words, each new run used both the original clauses of the problem and the shared clauses. Since the number of derived clauses is generally very large, copying and sharing all the derived clauses would swamp the original clauses and defeat the purpose of running many quick search runs. Thus we used a number of additional criterias for selecting the clauses to be shared.

In all the experiments we only picked clauses which were selected during the run as given clauses. The reason for this is that the heuristics of the given clause algorithm attempt to pick derived clauses which are more likely to lead to a proof. Although the heuristics are complex, the main focus is on preferring smaller or lighter clauses, with different measures of lightness.

The additional heuristics we have experimented with are clause length (less than 2, less than 3, unlimited) and the presence of variables. All the combinations were tried out, always in combination with sharing unit and two-literal clauses, using sequential runs without parallel processes. In these experiments the overall performance of the prover on the test set was slightly better than for the base case without sharing, but worse than for just sharing unit clauses. However, most of the experimental combinations managed to find a few proofs which were not found by the base case or unit sharing.

6 Summary and Conclusions

We have experimented with several options of sharing clauses between different runs of a portfolio strategy for proof search in first order logic formulas, implemented in the newest version of our high-performance prover GKC. The experiments show that it is worthwhile to share unit clauses between runs for simplification of derived and given clauses. In particular, this holds for both parallelized and non-parallelized runs. The experiment with 4 parallel processes and no limits on units shared gained 20 more proofs found over the non-sharing 294 proofs from the test set of 500 problems: not a huge gain, but significant. While gaining 36 proofs, sharing also lost 16 proofs. This indicates that a suitable heuristic strategy for controlling sharing might be able to lose fewer proofs and thus achieve even higher overall gains. Since the unit sharing and simplification algorithms used by GKC are relatively straightforward, it is likely that other provers would also gain from sharing unit clauses. As noted before, we have not tackled the issue of storing the parents of the shared clauses used during the cutoffs. Adding these to the shared memory would take additional time and space, the effects of which we cannot yet estimate. Since the gains diminish when parallelism increases, we assume that additional time spent on storing parent clauses would decrease the gains noticeably.

At the same time, we failed to find performance gains from additionally sharing two-literal clauses and full clauses. This said, we should not assume that these kinds of sharing are worthless. First, both the two-unit and full clause sharing produced several proofs which were not found in the non-sharing or unit-only sharing cases. It is not impossible that heuristics could be found which limit the sharing of longer clauses in a way that becomes beneficial. Second, the search strategies and algoritms used in different provers vary widely: we cannot guarantee

that sharing longer clauses would not benefit provers sufficiently different from GKC. Third, since GKC does not incorporate specialized propositional search algorithms, it could well be the case that sharing longer clauses specially for propositional handling like analyzed in [14] and [12] would benefit the propositional search components. For example, Rawson and Reger report in [12] experiments with a multithreaded Vampire with shared persistent grounding for propositional search, although their algorithms do not achieve overall gains on the test set.

References

[1] Avenhaus, J., Denzinger, J., Fuchs, M.: DISCOUNT: A System for Distributed Equational Deduction. In: Hsiang, J. (ed.) Proc. of the 6th RTA, Kaiserslautern. LNCS, vol. 914, pp. 397–402. Springer (1995)

[2] Bachmair, L., Ganzinger, H.: Resolution theorem proving. Handbook of automated reasoning **1**(02) (2001)

[3] Chvalovsky, K., Jakubuv, J., Olsak, M., Urban, J.: Learning theorem proving components. In: Automated Reasoning with Analytic Tableaux and Related Methods: 30th International Conference, TABLEAUX 2021, Birmingham, UK, September 6–9, 2021, Proceedings 30. pp. 266–278. Springer (2021)

[4] Denzinger, J.: Knowledge-Based Distributed Search using Teamwork. In: Proc. of the ICMAS-95, San Francisco. pp. 81–88. AAAI Press (1995)

[5] Denzinger, J., Kronenburg, M., Schulz, S.: 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

[6] Gleissa, B., Suda, M.: Layered clause selection for saturation-based. In: PAAR+SC@IJCAR. pp. 34–52. Springer (2020)

[7] Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: International Conference on Automated Deduction. pp. 299–314. Springer (2011)

[8] Kaliszyk, C., Urban, J., Vyskočil, J.: Lemmatization for stronger reasoning in large theories. In: Frontiers of Combining Systems: 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015, Proceedings 10. pp. 341–356. Springer (2015)

[9] Kinyon, M., Veroff, R., Vojtěchovskỳ, P.: Loops with abelian inner mapping groups: An application of automated deduction. Automated Reasoning and Mathematics: Essays in Memory of William W. McCune pp. 151–164 (2013)

[10] Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: International Conference on Computer Aided Verification. pp. 1–35. Springer (2013)

[11] McCune, W.: Release of prover9. In: Mile high conference on quasigroups, loops and nonassociative systems, Denver, Colorado (2005)

[12] Rawson, M., Reger, G.: A multithreaded vampire with shared persistent grounding. In: FMCAD. pp. 280–284 (2021)

[13] Reger, G., Suda, M., Voronkov, A.: Playing with avatar. In: Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25. pp. 399–415. Springer (2015)

[14] Reger, G., Tishkovsky, D., Voronkov, A.: Cooperating proof attempts. In: Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25. pp. 339–355. Springer (2015)

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

[16] Schulz, S., Möhrmann, M.: Performance of clause selection heuristics for saturation-based theorem proving. In: Olivetti, N., Tiwari, A. (eds.) Proc. of the 8th IJCAR, Coimbra. LNAI, vol. 9706, pp.

330–345. Springer (2016)

[17] Sutcliffe, G.: The CADE ATP system competition – CASC. AI Magazine **37**(2), 99–101 (2016)

[18] Tammet, T.: GKC: A reasoning system for large knowledge bases. In: Fontaine, P. (ed.) Proc. of CADE'2019 – the 27th Intl. Conf. on Automated Deduction. LNCS, vol. 11716, pp. 538–549. Springer (2019)

[19] Tammet, T., Draheim, D., Järv, P.: Gk: Implementing full first order default logic for commonsense reasoning (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) IJCAR 2022: Automated Reasoning. LNCS, vol. 13385, pp. 300–309. Springer (2022)

[20] Tammet, T., Järv, P., Draheim, D.: Confidences for commonsense reasoning. In: Platzer A., S.G. (ed.) Automated Deduction – CADE 28. CADE 2021. LNCS, vol. 12699, pp. 507–524. Springer (2021)

[21] Tammet, T.: Gandalf. Journal of Automated Reasoning **18**, 199–204 (1997)

[22] Tammet, T.: Towards efficient subsumption. In: International Conference on Automated Deduction. pp. 427–441. Springer (1998)

[23] Tammet, T., Järv, P., Verrev, M., Draheim, D.: An experimental pipeline for automated reasoning in natural language (short paper). In: International Conference on Automated Deduction. pp. 509–521. Springer (2023)

[24] TPTP homepage. http://www.tptp.org