Kalpa Publications in Computing

Volume 21, 2025, Pages 18–33

Proceedings of the 14th and 15th International Workshops on the Implementation of Logics

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 situation 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.

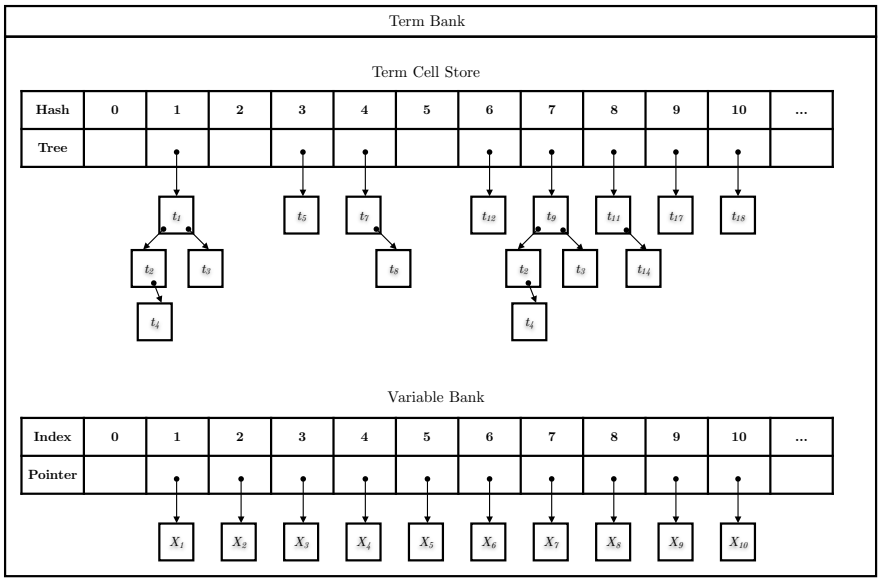

Figure 1: Term bank architecture

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 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 terms with the `f_code` field overloaded by the index value. De-Bruijn-variables come from a separate variable bank and 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 proceeds likewise, only with a different value of the garbage bit.

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

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

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.

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

[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.

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 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 $p(X)$.

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

- *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.

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

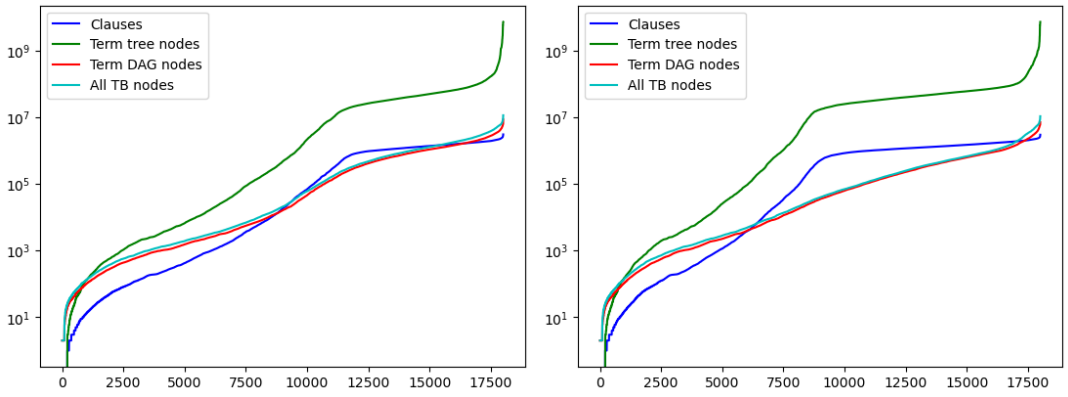

Figure 2: Distribution of clause number and different term nodes counts for auto-mode (left) and auto-mode without literal selection (right)

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 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-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.

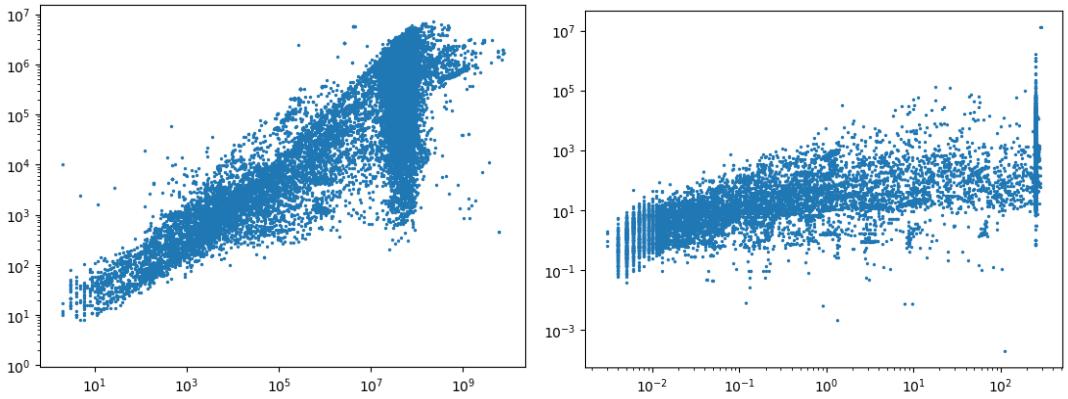

Figure 3: Scatter plot of term DAG nodes over term tree nodes for automatic mode (left) and of sharing factor over runtime (right). Notice that both axes are logarithmic for both diagrams.

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 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,

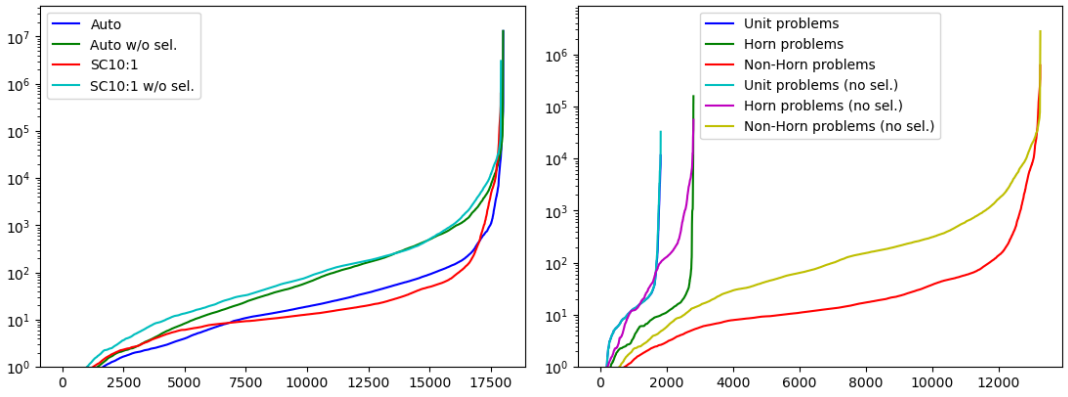

Figure 4: Distribution of the sharing factor for all four search strategies (left) and for Unit, Horn and non-Horn problems for the symbol counting strategy with and without literal selection (right)

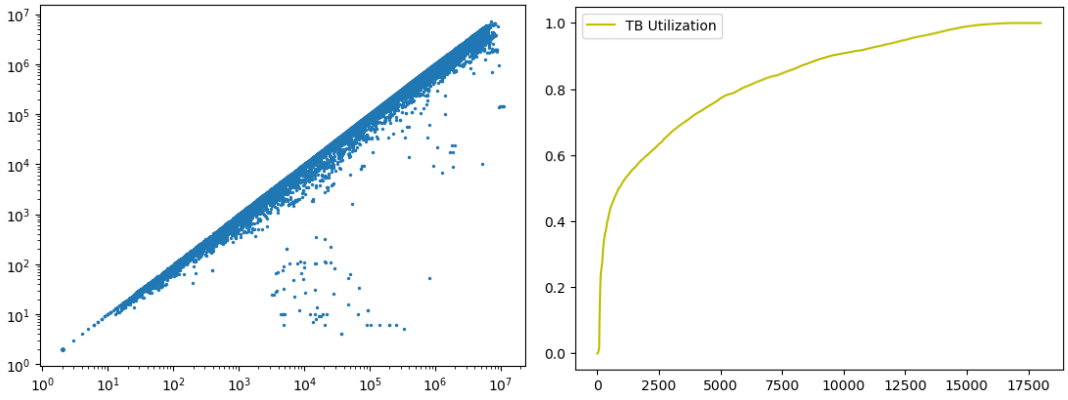

Figure 5: Referenced term bank nodes over all term bank nodes (left) and distribution of the utilization fraction (right) for automatic mode

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

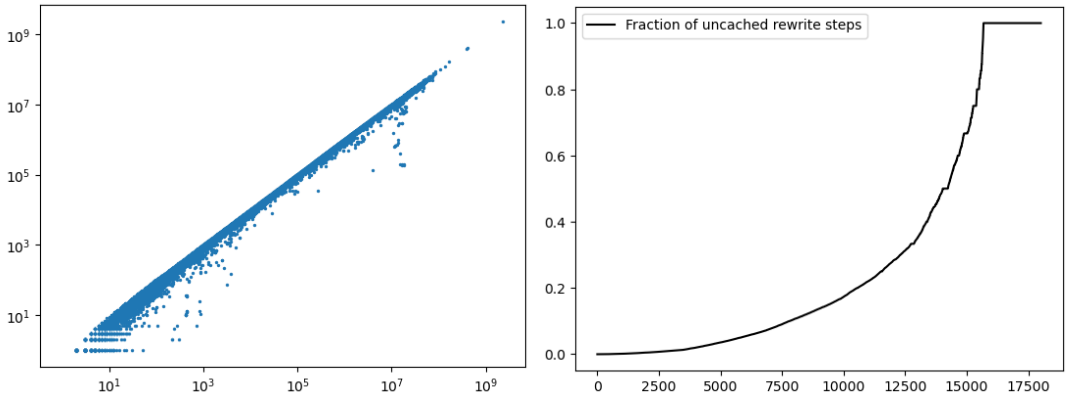

Figure 6: Cached rewrites over all rewrites (left) and distribution of the fraction of uncached rewrites (right) for automatic mode

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 logic [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.

In addition to the reduced memory usage and possible speed-ups, being able to delegate term and formula memory management to a garbage collector has increased developer productivity and reduced the number of memory leaks and pointer confusions.

Overall, a substantial amount of experience has been accumulated with shared terms and cached rewriting in E. We hope that this paper helps future implementations to avoid some of the pitfalls along the way, and to 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.