An Evaluation of Shared Rewriting

Bernd Löchner[1] and Stephan Schulz[2]

[1]FB Informatik, Universität Kaiserslautern, D-67663 Kaiserslautern, Germany, loechner@informatik.uni-kl.de
[2]Fakultät für Informatik, Technische Universität München, D-80290 München, Germany, schulz@informatik.tu-muenchen.de

Abstract

We present an experimental study on the use of shared rewriting in equational theorem proving. We identify the main effects that lead to term sharing in the proof state and experimentally show their influence. Besides the analysis of sharing and the *sharing factor* we gain insights on the benefits for rewriting and matching operations by a comparison of the theorem provers E and Waldmeister. This also allows us to conclude that the influence of the side effects of E's use of shared rewriting on the search process is quite minor.

1 Introduction

The thorough understanding of techniques found in current theorem provers is a prerequisite for the creation of more powerful new systems. Especially the potential of unusual approaches is often not well understood and, due to the complexity of the issue, eludes theoretical analysis.

Shared term data structures represent repeated instances of the same subterm in the proof state only once. Such representations have been in use for a number of years. However, we are not aware of any more than cursory analysis of the behavior of shared representations. In addition to the obvious benefit of reducing the memory consumption (the original motivation for shared representations), they also allow completely new techniques. Shared, non-local rewriting on shared terms, as realized in the theorem prover E [Sch01], is such a technique.

Most high-performance theorem provers use indexing techniques to replace many operations working with an element at at time with one operation working with a whole set at at time. This is typically used for matching and (less often) unification. Instead of sequentially trying one clause after the other, an index on the clause set is used to quickly determine a set of candidate clauses for the desired operation. Shared rewriting applies the *set-at-a-time* concept to a new class of operations, namely to term rewriting and normal form computation, by rewriting all instances of a given term at the same time.

In this paper we analyze the behavior of E's implementation of shared rewriting and compare it with the more conventional techniques found in Waldmeister [HJL99]. This system was chosen because both provers have been developed using experiences gained with the DISCOUNT system [DKS97] and are therefore similar in structure: Both use the DISCOUNT loop variant of the given clause algorithm, both use variants of perfect discrimination trees as an indexing structure to speed up most unit operations, and both use similar heuristics to control the proof search.

Despite this similarity the two provers are built on very different core data structures. Waldmeister has been designed around a variant of perfect discrimination trees [McC92] with the aim of maximal efficiency in terms of memory and time. Consequently, its term representation is a flat term structure [Chr93] optimized for fast traversal and memory efficiency. E, on the other hand, is built around perfectly shared terms. Whole term sets are represented by a single directed acyclic graph, and all shared instances of a term are rewritten at once. To our knowledge, E is the only prover that performs rewriting directly on a shared term representation.

Both systems are among the most powerful provers for problems formulated in unit-equational logic as demonstrated at the recent CASC-JC [Sut01, SSPar]. Among general purpose provers, E enjoys a significant lead over the competing systems, and Waldmeister, which is specialized for unit-equality problems, has an approximately similar lead over E.

This paper is structured as follows: In section 2 we introduce the necessary background. The core of the paper is formed by sections 3 and 4, which analyze the static and dynamic aspects of the shared term representation, respectively. We conclude in section 5.

2 Unfailing Completion and Equational Theorem Proving

E and Waldmeister are based on different, but related calculi. E is a superposition [BG94] prover for full clausal logic with optional selection of negative literals. Waldmeister is based on unfailing completion [BDP89], extended with narrowing to be able to deal with existentially quantified variables in the goal. Since non-ground goals are handled differently by both provers, we restrict our discussion to the case of unit-equational theories with a single unit ground goal. For this case the superposition calculus degenerates into unfailing completion.

2.1 Proof search organization and heuristics

Unfailing completion is a *saturating* calculus, i. e. it systematically enumerates consequences of the input clauses. Most current high-performance theorem provers are based on such calculi, and most of those use a variant of the *given-clause algorithm* to achieve this enumeration. In this algorithm the set of all clauses is partitioned into a set $\mathcal{A}$ of *active* (or *processed*) clauses and a set $\mathcal{P}$ of passive (or *unprocessed*) clauses. Initially, $\mathcal{A}$ is empty and $\mathcal{P}$ contains the input. During each traversal of its main loop the algorithm picks a clause c (the *given clause*) from $\mathcal{P}$. This selection is controlled by an *evaluation heuristic* φ, which typically prefers small or old clauses. If c cannot be shown to be redundant, the algorithm inserts c into $\mathcal{A}$, generates all direct consequences between $\mathcal{A}$ and c, and finally adds them to $\mathcal{P}$. In this case we say that c has been *activated.* Another important parameter of the algorithm is the *reduction ordering* $>$ which can have a strong influence on the proof search.

The given-clause algorithm is used in two major variants, the *DISCOUNT loop* and the *Otter loop*, both named after the system that made the corresponding version popular. The difference lies in the treatment of simplification. In the DISCOUNT variant only the given clause and the active clauses are used for simplification. Newly generated clauses are simplified once after generation, and again if they are picked for processing. Active clauses are back-simplified with the given clause. Passive clauses, as their name implies, are not used for simplification. This leads to a very high inference rate and allows a variety of optimizations. In particular, since passive clauses do not participate in any inferences, they can be stored very efficiently[1]. Furthermore, the so-called orphan-criterion applies: A clause can be deleted if one of its "parents" has been interreduced.

Algorithm 1 The proof procedure of WALDMEISTER

FUNCTION WALDMEISTER($\mathcal{I}np, >, \varphi, strength$) : BOOLEAN
$(\mathcal{A}, \mathcal{P}, \mathcal{H}) := (\varnothing, \text{axioms}(\mathcal{I}np), \text{hypotheses}(\mathcal{I}np))$
WHILE $\neg\text{trivial}(\mathcal{H}) \wedge \mathcal{P} \neq \varnothing$ **DO**
    $e := \arg\min \varphi(\mathcal{P});\ \mathcal{P} := \mathcal{P} \setminus \{e\}$
    IF $\neg\text{orphan}(e)$ **THEN**
        $e := \text{Normalize}_{\mathcal{A}}^{>}(full, e)$
        IF $\neg\text{redundant}(e)$ **THEN**
            $(\mathcal{A}, P_1) := \text{Interred}^{>}(\mathcal{A}, e)$
            $\mathcal{A} := \mathcal{A} \cup \{e\}$
            $P_2 := \text{Superpos}^{>}(e, \mathcal{A})$
            $\mathcal{P} := \mathcal{P} \cup \text{Normalize}_{\mathcal{A}}^{>}(strength, P_1 \cup P_2)$
            $\mathcal{H} := \text{Normalize}_{\mathcal{A}}^{>}(\mathcal{H})$
        END
    END
END
RETURN $\text{trivial}(\mathcal{H})$

Both E and Waldmeister use this organization as can be seen in Alg. 1 and Alg. 2 respectively. The main (conceptional) difference between the provers lies in the handling of hypotheses. Waldmeister keeps the hypotheses in a separate set $\mathcal{H}$ whereas in E there is no special treatment. So the test predicate for

[1] In Waldmeister passive clauses are either stored in a compressed form or are entirely deleted and reconstructed by need.

Algorithm 2 The proof procedure of E (simplified)

FUNCTION EPROVER($\mathcal{I}np, >, \varphi, strength$) : BOOLEAN
1: $(\mathcal{A}, \mathcal{P}, P_0) := (\varnothing, \mathcal{I}np, \varnothing)$
2: **WHILE** $\Box \notin \mathcal{A} \cup P_0 \wedge \mathcal{P} \neq \varnothing$ **DO**
3: $\quad c := \arg\min \varphi(\mathcal{P});\ \mathcal{P} := \mathcal{P} \backslash \{c\}$
4: $\quad$ **IF** $\neg$orphan(c) **THEN**
5: $\quad\quad c := \text{Normalize}^{>}_{\mathcal{A}}(full, c)$
6: $\quad\quad$ **IF** $\neg$redundant(c) **THEN**
7: $\quad\quad\quad (\mathcal{A}, P_1) := \text{Interred}^{>}(\mathcal{A}, c)$
8: $\quad\quad\quad \mathcal{A} := \mathcal{A} \cup \{c\}$
9: $\quad\quad\quad P_2 := \text{Superpos}^{>}(c, \mathcal{A})$
10: $\quad\quad\quad P_0 := \text{Normalize}^{>}_{\mathcal{A}}(strength, P_1 \cup P_2)$
11: $\quad\quad\quad \mathcal{P} := \mathcal{P} \cup P_0$
12: $\quad\quad$ **END**
13: $\quad$ **END**
14: **END**
15: **RETURN** $\Box \in \mathcal{A} \cup P_0$

a successful proof are slightly different as well: E detects the generation of the empty clause $\Box$, Waldmeister tests whether the simplification of $\mathcal{H}$ joins a hypothesis to the trivial equation $t = t$. Besides that, the systems are very similar. As an example, both systems offer a parameter *strength*. This allows to specify how much effort shall be spent in the normalization of newly generated clauses (line 11 in Alg. 1 and Alg. 2), since most of the time in the proof process is consumed by this routine. Of course, the simplification at activation time (line 5) is always done with full strength.

In the Otter loop the simplification relation is induced by all clauses. Implementations vary in detail, but typically all unit clauses are used for both forward and backward rewriting. Thus, redundant clauses can be removed or simplified earlier. However, this implies that all clauses need to be stored in a way that makes them available for inferences. In particular, if indexing techniques are used, *all* clauses have to be indexed, and all clauses have to be stored explicitly. Furthermore, the orphan-criterion is not applicable.

2.2 The implementation of shared rewriting

All terms in E are stored in a *term bank*. They are inserted in a bottom-up fashion, i. e. a term's key in the term bank is computed from the function symbol and the list of pointers to its already shared arguments. The term bank itself is organized as a large hash table to allow the fast access of a term node via its key. Each term node carries, besides its key, information about its superterms and about external references. Term rewriting replaces a term cell wherever it is referenced and recursively traverses all superterms to ensure that the term bank has again only one unique node for each represented term. Hence, any single rewrite step potentially affects a large number of clauses. In addition to the reference pointers the term nodes also carry various information, e. g. about the time they were last rewritten to normal form.

Clauses are represented as lists of equational literals and contain various precomputed values. Passive clauses are associated with an arbitrary number of *heuristic evaluations* and are entered in an equivalent number of priority queues. Due to this very general design unit clauses carry a significant size overhead compared with Waldmeister's more specialized data structures.

The indexing data structure used in E is a perfect discrimination tree with age constraints. Each branch in the index is annotated with the age of the youngest clause it contains. These annotations and the normal form dates stored in each term cell enable the matching routine to cut off entire branches of the discrimination tree early if no useful match is possible. We found that for reasonably hard examples, this saves about 10% of overall time and about 30% of the time for matching and ordering tests for rewriting.

3 Experiments concerning the sharing factor

The benefits of the shared term approach obviously depend on the amount of sharing achieved, i. e. on the average number of standard term nodes each shared term cell represents. We call the ratio between the number of subterms in a collection of terms (in our case usually all terms in the proof state) and the shared terms nodes necessary to represent them the *sharing factor*. In this section, we analyze the behavior of E with respect to this central parameter.

There are several possible sources for the duplication of terms in the search state of a superposition based theorem prover. Typically, even in the initial clause set, small terms like variables or constant terms occur more than once. An analysis of the calculus reveals a number of potential reasons for an increase in the number of duplicated terms during the inference process. Ignoring the ordering restrictions, the superposition inference rule[2] can be expressed as follows (we write $u \dot{\simeq} v$ to cover both $u \simeq v$ and $u \not\simeq v$):

$$\frac{s \simeq t \vee C \quad u \dot{\simeq} v \vee D}{\sigma(u \dot{\simeq} v[t]_p \vee C \vee D)} \quad \text{where } \sigma = \mathrm{mgu}(s, v|_p)$$

As we can see, nearly all (sub)terms of the new clause are instantiated copies of terms from the parent clauses. This introduces duplication via three mechanisms.

1. The parts of the parent clauses containing no or only unbound variables are not affected by σ and are copied verbatim, leading to sharing between the new clause and the old ones.

2. The application of σ leads to sharing within the new clause whenever a bound variable occurs more than once in the uninstantiated new clause.

[2]The additional generating inference rules necessary for the non-unit case are applied very rarely in practice and are therefore insignificant for this analysis.

3. Often σ contains *copying bindings*, i. e. $\sigma(x)$ is an uninstantiated subterm from one of the parent clauses. This again leads to sharing between this parent and the new clause.

These considerations lead to the following hypothesis:

Hypothesis 1 The average sharing factor increases during the saturation process and is correlated to the size of the term set. Extremely high degrees of sharing are characterized by the presence of highly non-linear or permutative clauses like e. g. the AC axioms.

Experiment 1 The theorem prover E is retrofitted to determine the number of shared term cells and the number of standard cells they represent.

To establish enough diversity and therefore allow general conclusions, we use a large number of specifications with different basic parameters such as difficulty, size and structure of the signature, or number of axioms. The sharing factor is determined at the end of each run, when E has either solved the problem or has encountered a given resource limit. We use the set of all unit-equational problems with ground goals from TPTP 2.4.1 [SS98] as test problems, i. e. 380 proof tasks[3]. We use two different clause selection heuristics (symbol counting and symbol counting interleaved with age-based selection with a pick-given ratio of 5:1) and two term orderings, a lexicographic path ordering (LPO) and a Knuth-Bendix Ordering (KBO). We also use two different strengths for rewriting newly created clauses. Full rewriting uses all processed unit clauses to rewrite the new clauses. Alternatively, we perform rewriting with oriented units only, thus saving the potentially costly ordering tests of the instantiated equations. The combination of the three parameters results in 8 different parameter settings.

Additionally, we analyze selected examples in detail. At each iteration of the main saturation loop the given clause is recorded and the sharing factor is measured. This allows us to correlate the development with the structure of the given clauses selected and thus to partially explain the observations.

All tests were performed on SunBlade 1000 computers with 750 MHz processors, a time limit of 500 seconds and a memory limit of 192 MB.

Result 1 Figure 1 shows the sharing factor at the termination of the prover over the number of clauses generated during the run. The number of generated clauses is a good indicator for the amount of work done by the prover, and in most cases is strongly correlated to the total number of terms in the proof state. The average sharing factor increases from about 1 for trivial examples to over 15 for larger ones. About 5% of the runs show a sharing factor higher than 20. The diagram does not include a small number of outliers with values between 60 and 3200. There is no indication that search parameters or reduction ordering have a significant influence on the sharing factor.

[3] Waldmeister and E handle non-ground goals very differently, and no useful comparison is possible for this case.

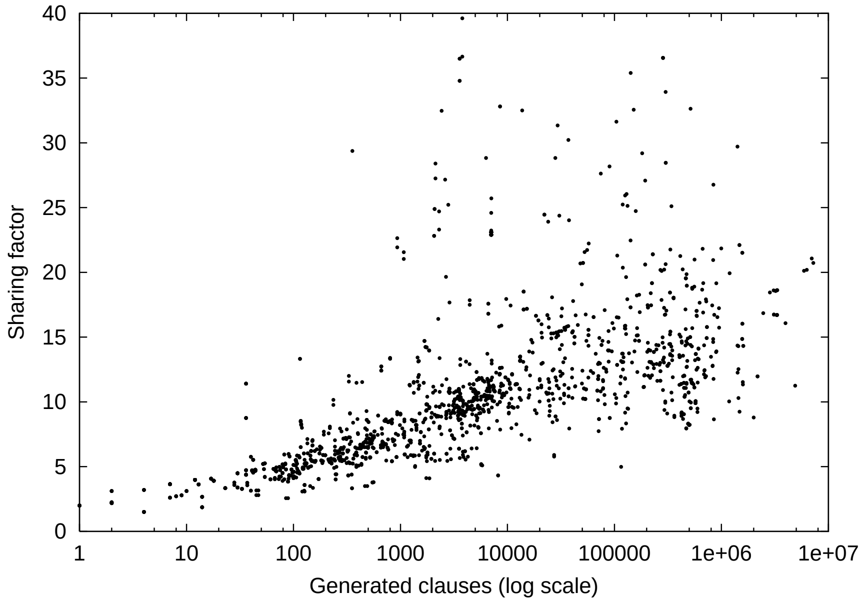

Figure 1: The sharing factor as a function of the length of the runs (measured as number of generated clauses).

The outliers are problems from group theory and from the ROB domain in TPTP. A detailed analysis of individual examples shows that the most extreme cases of sharing occur for problems with highly non-linear axiomatizations. For the problem GRP207-1 we observe variables with more than 30 occurrences in the right hand side of an equation after a very moderate number of activations.

A more interesting pattern can be observed if permutative equations, in particular AC axioms, are present. Figure 2 shows the development of the sharing factor for ROB005-1, a typical AC specification of medium difficulty. We can identify three periods with a strong monotonic increase of the sharing factor. During each of these periods all activated clauses belong to a new class of permutative equations. In these periods the sharing factor is increased due to the third mentioned mechanism: Overlaps with permutative equations result in substitutions with copying bindings only. If E's AC redundancy elimination mechanism (based on [AHL00]) is activated, the graph shows a much more stable behavior, with a sharing factor in the more moderate range of about 10 for most of the time. Thus, the high sharing factor is an indicator for the presence of a large number of very similar redundant clauses.

3.1 Sharing in the non-unit case.

To find out how these results generalize to the non-unit case we also performed experiments with Horn and non-Horn specifications. Due to the much larger

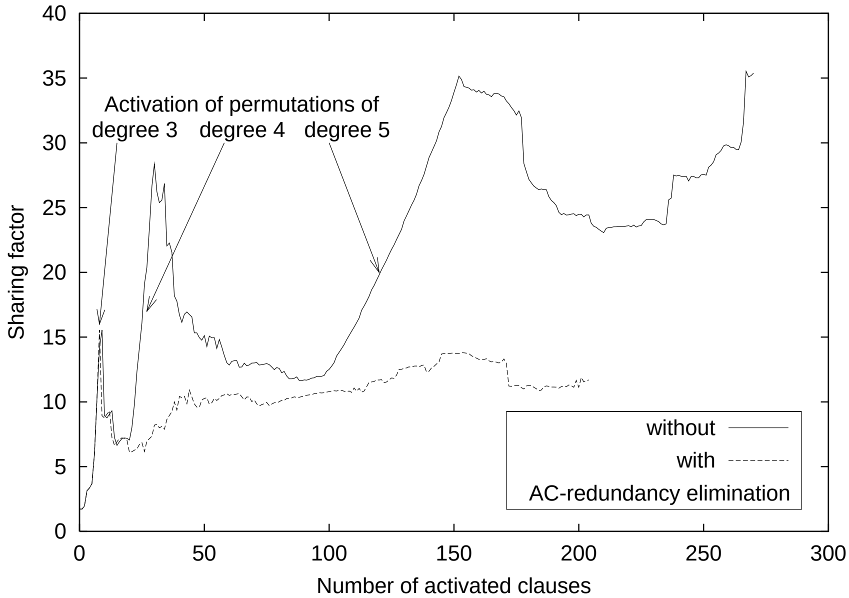

Figure 2: Development of the sharing factor over activated clauses for ROB005-1

amount of such problems contained in the TPTP, we were not able to perform the experiments as thoroughly as for the unit case. However, we performed two sets of runs, one without literal selection and one with literal selection. Both were run on SUN Ultra-10/300MHz machines, with a time limit of 300 seconds. This is roughly equivalent to 125 seconds on the faster machines used for the unit experiments.

For many problems with non-Horn specifications the sharing factor is very high – more than 100 for about 30% and about 10000 for roughly 10% of the problems. Generally, using literal selection leads to a somewhat lower sharing factors.

In the Horn case we observe a very interesting effect. If we run the prover with literal selection, which results in a positive unit strategy, the bulk of examples has sharing factors from 5 to 15. This is very similar to the unit case. Without literal selection the behavior is similar to the non-Horn case: Many examples exhibit a sharing factor of about 50, and a still significant number a factor of more than 200.

We interpret this results as follows: As we stated in the analysis at the beginning of section 3, there are three causes for the creation of shared terms: Copying of the unmodified context of an inference, instantiation of non-linear variables, and copying variable bindings. We believe that the first two effects are primarily responsible for the behavior in the non-unit case.

If we perform a generating inference between two non-unit clauses, we suffer from the so-called *duplication problem* [LP92]. All literals of the precondition clauses except for the inference literals are copied (in instantiated form) into the conclusion. If the clauses have more than two literals, this leads to an exponential growth in the number of literals in a clause. Thus, non-unit inferences lead to very large clauses – and inferences with large clauses lead to a lot of duplicated terms, both because there is more context copied in each inference, and because, on average, the degree of non-linearity increases.

In the Horn case, literal selection leads to a unit-strategy, i. e. no inferences involving more than one non-unit clause are performed, and hence the length of the clauses never increases. This explains why in this case the sharing factor behaves very similar to the unit case. Without literal selection, arbitrary inferences are possible, and thus we see both longer clauses and more sharing. As an example, consider the problem `BOO006-1` from TPTP, a problem of medium difficulty. If we use the simple clause weight heuristic without literal selection, the average number of literals per clause, measured at the time the total number of clauses in the proof state reaches 20000, is approximately 3.4, and the sharing factor is 475. With strict literal selection (always select the smallest negative literal), that numbers drop to 1.26 for the literals per clause and 78 for the sharing factor.

In the non-Horn case, literal selection no longer results in a unit strategy. However, it still results in positive strategy, and thus blocks all inferences between clauses with both positive and negative literals. Most long clauses have both positive and negative literals, and hence literal selection is able to limit the growth of clauses to some extend. Again, we look at an example of medium difficulty, `SWC398-1`. Without literal selection, the average literal number per clause is 5.4, and the sharing factor 244. With literal selection, we have 1.21 literals per clause, and a sharing factor of 6.2. As can be seen in both cases, a higher literal count (implying, on average, more inference context), also leads to a higher sharing factor.

4 Experiments concerning the shared rewriting

After analyzing the influence of the shared term implementation on the *static* representation of each consecutive search state, we will now discuss the effects of term sharing on more *dynamic* aspects, i. e. on the rewriting and matching operations and on the influence on the search behavior.

4.1 Influence on the search behavior

Comparing the number of rewrite steps performed by a system with shared rewriting with the number performed by a standard implementation is surprisingly hard. A first, conceptually simple, approach would be to implement both forms of rewriting in a single system, and to compare the two results. However, the shared rewriting has a profound influence on the design of many different parts of a proof system. Adapting all these parts for the conventional case amounts to reimplementing large parts of the prover. This is not feasible within reasonable resources.

Alternatively, one could analyze the behavior of E, and attempt to figure out how many conventional rewrite steps correspond to each shared inference performed by E. However, this is very hard: Due to sharing, simplification steps performed with new clauses affect as a side effect clauses in $\mathcal{P}$ a conventional prover would never touch. Furthermore, we gain no insights into the influence of shared rewriting on search behavior and running time.

We finally settled on the the following approach: Since E and Waldmeister are similar in their structure (except for shared rewriting!) we adjust them to behave as similar as possible. If this is achievable for a significant number of diverse problems, we can compare the number of conventional rewrite steps performed by Waldmeister with the number of shared steps performed by E.

Hypothesis 2 It is possible to make E and Waldmeister behave similar for a wide variety of proof tasks by adjusting search parameters of both systems, i. e. the introduction of shared rewriting does not significantly alter the search behavior.

Experiment 2 For both provers we fix the clause evaluation heuristic, the rewrite relation for forward simplification of newly generated clauses, the rewrite strategy (leftmost-innermost), and the term ordering to identical settings. We also disable techniques like AC-redundancy elimination or special goal handling that are only implemented in one prover or are likely to lead to significant differences.

We then compare the behavior of the provers for corresponding strategies. A good indicator for the similarity of two runs is the comparison of the number of clauses generated until a proof is found. Due to the known instability of most saturating proof techniques this number normally varies greatly for even minor differences in the search process.

We use the same problem set and machine configuration as in Experiment 1. Since both systems vary in inference speed it is not sensible to compare runs when one or both encounter a timeout. We therefore restrict our attention to the 983 runs where both systems find a proof within the given resource limits of 500 seconds and 192 MBytes.

Result 2 Figure 3 shows the number of steps performed by Waldmeister over the number of steps performed by E for each proof problem successfully proved by both systems. As we can see, a large majority of examples shows very similar behavior for both systems. If we allow a difference of 20% only, 630 out of 983 runs, i. e. 64% of all runs, are similar. Outliers are distributed quite evenly on both sides. Considering the above mentioned instability both provers show a remarkably similar behavior.

Thus, contrary to our initial expectation, we find no strong evidence that the choice of the term representation introduces a qualitatively different behavior in the unit-equational case.

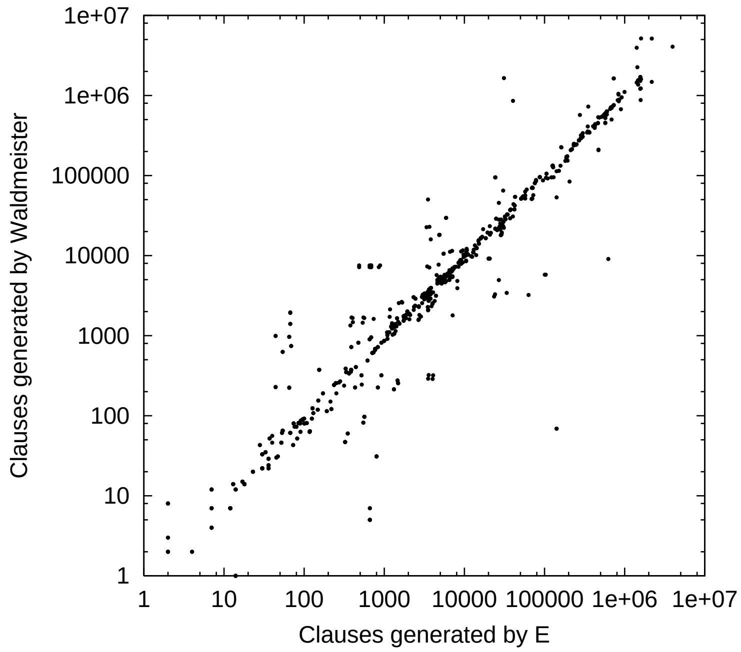

Figure 3: Clauses generated by Waldmeister over clauses generated by E for 983 runs

4.1.1 Stability of the clause count.

Our hypothesis, that even small disturbances in the inference process lead to vastly different numbers of generated clauses, is demonstrated by a different set of experiments. We also ran E and Waldmeister with an interleaved *symbol counting/first-in-first-out* heuristic. In this case, E counts activations of the goal as part of the clause selection process, while Waldmeister not, since it treats the goal as a distinct object. This small difference leads to a much wider spread of results. In this case, only 21% of the runs lie in the 20% difference interval.

4.2 The number of performed rewrite steps

For the influence of shared rewriting on normal form computations two main effects have to be considered. First, each rewrite step in E corresponds to multiple conventional rewrite steps, since all instances of the rewritten term are replaced at once. Second, the normal form procedure stores knowledge about previous match attempts in the term node and thus avoids redundant match attempts. This generalizes a typical optimization in innermost rewriting, namely the marking of irreducible subterms.

Hypothesis 3 The ratio between non-shared and shared rewrite steps and

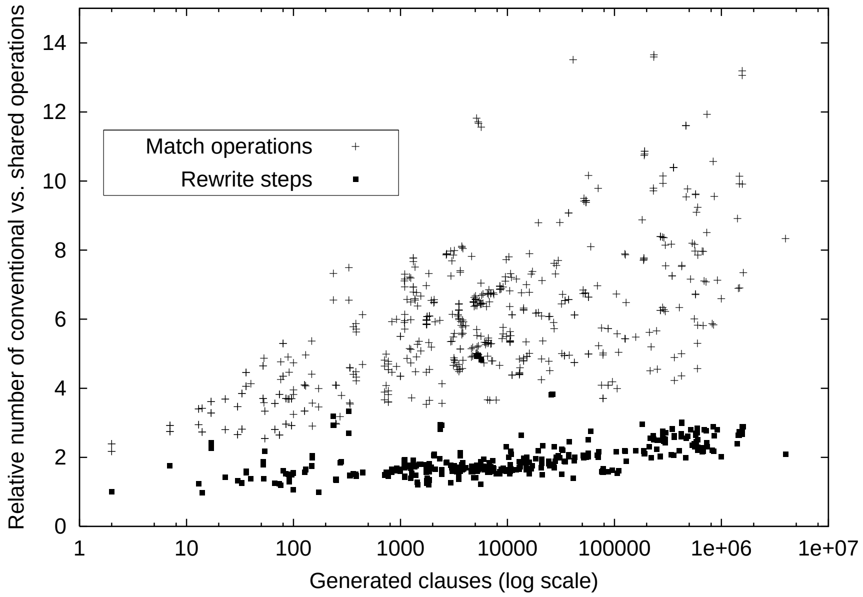

Figure 4: Quotient of conventional vs. shared rewriting steps resp. matching operations

matching operations increases for longer runs. This behavior is more pronounced for matching operations.

Experiment 3 We compare the number of rewrite steps and match operations performed by Waldmeister and E for the similar runs, i. e. where the number of generated clauses differs by at most 20% between both provers, determined in Experiment 2.

Result 3 Figure 4 shows both the relative proportion of unshared rewrite steps performed by Waldmeister versus shared rewrite steps performed by E and the relative proportion of matches attempted by both provers. Both values have been corrected for the small differences in the number of generated clauses.

For the rewrite steps we see a quite homogenous result, with the ratio of unshared to shared inferences rising from about 1 for easy problems to about 2.5 for harder problems. For the number of matches the spread of results is much bigger. The ratio of match attempts rises from 1 to about 9, with the spread for harder problems going from about 5 to about 14. A small number of outliers display a ratio of 4 or 5 in rewrite steps and a ratio from 15 to 25 for match attempts.

As expected, the ratio of conventional vs. shared operations rises with time. The small effect for rewrite steps is explained by the fact that the vast majority originates from normalizing the newly generated clauses. The number of such clauses typically only grows moderately over time. Most sharing, on the other hand, occurs in the much larger set $\mathcal{P}$. Since most clauses in $\mathcal{P}$ are never activated, the implicit rewriting of terms in these clauses has no counterpart in Waldmeister. For match attempts the normal form information at each shared note is crucial. If E encounters a term that has been normalized with the current rewrite relation, it can avoid matches on this term *and* on all of its subterms.

4.3 Comparison of the run times

Up to now, we have only measured and compared quantities that are implementation independent. This allows to asses the properties of a method or technology regardless of peculiarities of actual implementations. When comparing the run times of E and Waldmeister we have to take into account the disturbances due to different levels of optimizations in the actual code. Nevertheless, analyzing the ratio of run times over the problem complexity, that is, the time needed to solve it, gives insights into the suitability of shared rewriting for challenging proof tasks.

Hypothesis 4 There are no fundamental differences between shared and non-shared rewriting over the run time. Shared rewriting can cope better than non-shared rewriting with the additional costs for full simplification of newly generated clauses compared to simplification with oriented units only.

Experiment 4 We compare the run times of Waldmeister and E for the similar runs, i. e. where the number of generated clauses differs by at most 20% between both provers, determined in Experiment 2.

Result 4 As expected, Waldmeister is faster in most cases. But we find no significant correlation between the ratio of run times and the ratio of matching operations or rewrite steps.

An interesting pattern concerning the strength of initial simplification can be observed: If only orientable units are used, Waldmeister is about four to five times faster than E, independent of the magnitude of the run time. If full simplification is used, Waldmeister is only 2.5 times faster on the average, the variance is much higher, and there are even some examples, where E is faster than Waldmeister. Here too, no significant change over the run time is observed.

There are six runs where E is faster than Waldmeister and Waldmeister needs more than ten seconds. With one exception, they all belong to problems with one or two AC-symbols in the axiomatization. For Waldmeister, restricting the strength of the simplification relation is especially beneficial for this class of problems. For E, the effect is not so pronounced – in fact, in our experience E performs better with the full strenght rewrite relation on nearly all problems. This is consistent with the above findings.

Rewriting with orientable instances of unorientable equations, as done in the case of the full strength rewrite relation, requires an ordering comparison after each successful matching operation. This test leads to an increased costs for

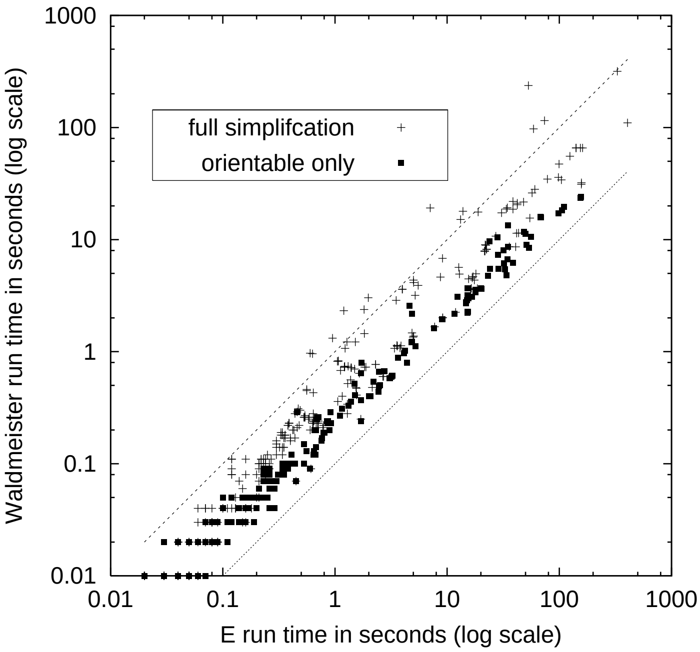

Figure 5: Comparison of actual running times of E and Waldmeister. The two boundary lines correspond to the case of equal speed for both provers and to the case where Waldmeister is 10 times faster. Note that the scales are logarithmic.

rewrite attempts with such equations. In the case of AC-symbols the following effect can be observed: The commutativity axiom matches at all positions where its function symbol occurs. Then the ordering constraints have to be checked. Most of the time they are not fulfilled. The situation is similar with other small permutative equations. As a result the normalization relation becomes notably more expensive. It seems that in this situation the normal form dates stored in the term nodes are especially profitable for E. We conclude that shared rewriting has advantages when the rewrite relation is costly.

5 Conclusion

Our analysis shows to our surprise that for unit equality problems there is no evidence for a significant difference caused by the choice of the term structure. The memory saving possible with sharing can also be achieved by much simpler compression techniques. Concerning rewriting, the positive effects of slightly fewer rewrite steps and noticeably fewer matches seem to be neutralized by the much higher cost for rewriting and term bank administration. When the rewrite relation is more expensive, the situation changes a little bit, as the analysis of the overall run times shows. There is some evidence that E can deal better with search situations with a high degree of redundant information. We also find that the shared term representation in E masks a number of weaknesses in the inference engine. In particular, term comparisons with KBO and LPO are much slower in E, and E does not use certain optimizations in indexing symmetric unit clauses. This seems to be compensated by the normal form dates in the term nodes.

Nevertheless, the shared rewriting paradigm requires a significantly higher implementation effort to get a working prover, and for the unit-equational case development resources should probably be invested into other areas.

We conjecture that shared rewriting promises more benefits for other settings. First, for provers using the Otter loop which have to represent all clauses explicitly, the space savings caused by the shared term representation can become important. Secondly, since the shared term implementation can save a significant amount of match operations, it should pay off in the case where matching is more expensive, e. g. in the case of AC rewriting. Finally, the much higher sharing factors observed in the non-unit case indicate that the real potential of the approach lies in this area. Unfortunately, we currently see no possibility of a detailed comparison for this case, since no prover with a structure sufficiently similar to E is known to us.

This study has had one positive side effect: Both authors have gained a far better understanding of both systems and of the implemented techniques. We found some minor bugs in one of E's heuristics and in some reporting code.

The outcome of these experiments has given us new insights about where to invest development effort. For example, the sharing factor seems to be a good indicator for the behavior of the search heuristic: A high sharing factor indicates that the prover generates a lot of redundant clauses and should probably change its search behavior.

References

[AHL00] J. Avenhaus, T. Hillenbrand, and B. Löchner. On Using Ground Joinable Equations in Equational Theorem Proving. In P. Baumgartner and H. Zhang, editors, *Proc. of the 3rd FTP, St. Andrews, Scottland*, Fachberichte Informatik. Universität Koblenz-Landau, 2000. (revised version to be published in the Journal of Symbolic Computation).

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

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

[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.

[HJL99] T. Hillenbrand, A. Jaeger, and B. Löchner. System Abstract: Waldmeister – Improvements in Performance and Ease of Use. In H. Ganzinger, editor, *Proc. of the 16th CADE, Trento*, volume 1632 of *LNAI*, pages 232–236. Springer, 1999.

[LP92] S.-J. Lee and D.A. Plaisted. Eliminating Dupliction with the Hyper-Linking Strategy. *Journal of Automated Reasoning*, 9(1):25–42, 1992.

[McC92] W.W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[Sch01] S. Schulz. System Abstract: E 0.61. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proc. of the 1st IJCAR, Siena*, volume 2083 of *LNAI*, pages 370–375. Springer, 2001.

[SS98] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[SSPar] G. Sutcliffe, C.B. Suttner, and J. Pelletier. The IJCAR ATP System Competition. *Journal of Automated Reasoning*, to appear.

[Sut01] G. Sutcliffe. The CASC-JC Web Site. `http://www.cs.miami.edu/~tptp/CASC/JC/`, 2001.