The Next Waldmeister Loop

Thomas Hillenbrand[1] and Bernd Löchner[2]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany,
hillen@mpi-sb.mpg.de
[2] FB Informatik, Universität Kaiserslautern, Kaiserslautern, Germany,
loechner@informatik.uni-kl.de

Abstract. In saturation-based theorem provers, the reasoning process consists in constructing the closure of an axiom set under inferences. As is well-known, this process tends to quickly fill the memory available unless preventive measures are employed. For implementations based on the Discount loop, the passive facts are responsible for most of the memory consumption. We present a refinement of that loop allowing such a compression that the space needed for the passive facts is linearly bound by the number of active facts. In practice, this will reduce memory consumption in the Waldmeister system by more than one order of magnitude as compared to previous compression schemes.

1 Introduction

Theorem proving procedures are refutationally complete under the idealized assumption of unlimited resources. But implementations thereof are run on finite machines and so can only analyze a finite part of the infinite search space. For provers based on saturation procedures, this resource restriction shows up as sky-high growing memory consumption if, in order to solve a difficult problem, they need to be run not for a few minutes only, but for say hours to days; and a bundle of preventive measures thereagainst has been suggested.

The Waldmeister system is an example of such an implementation. Since the very beginning, ad-hoc compression techniques have been employed, condensing the individual conclusions derived into a more compact representation. Spurred by the renewed discussion of how to organize the proof search within saturation-based theorem provers – Otter or Discount loop [Vor01, pp. 15–17] – we reconsider our system's proof procedure in this paper (Sect. 2) and show that the latter loop variant in an elegant way allows the compression of coarse-grained sets of conclusions (Sect. 3.1). Our technique requires for completeness reasons that the successive normal form functions developed during completion remain accessible. We present a simple and efficient realization of this requirement in Sect. 3.2. Since we avoid to discard non-redundant critical pairs, the prover delivers a decision procedure in case the saturation process terminates.

The Discount loop thereby being developed into its ultimate form, memory consumption will be reduced by more than one order of magnitude with this concept, which we demonstrate by a simulation-based estimation (Sect. 4). Today's

Algorithm 1 The proof procedure of WALDMEISTER

FUNCTION WALDMEISTER$(\mathcal{E}, \mathcal{C}, >, \varphi)$: BOOLEAN
$(\mathcal{A}, \mathcal{P}) := (\varnothing, \mathcal{E})$
WHILE $\neg\text{trivial}(\mathcal{C}) \wedge \mathcal{P} \neq \varnothing$ **DO**
  $e := \min_\varphi(\mathcal{P});\ \mathcal{P} := \mathcal{P} \setminus \{e\}$
  IF $\neg\text{orphan}(e)$ **THEN**
    $e := \text{Normalize}_{\mathcal{A}}^{>}(e)$
    IF $\neg\text{redundant}(e)$ **THEN**
      $(\mathcal{A}, P_1) := \text{Interred}^{>}(\mathcal{A}, e)$
      $\mathcal{A} := \mathcal{A} \cup \{\text{Orient}^{>}(e)\}$
      $P_2 := \text{CP}^{>}(e, \mathcal{A})$
      $\mathcal{P} := \mathcal{P} \cup \text{Normalize}_{\mathcal{A}}^{>}(P_1 \cup P_2)$
      $\mathcal{C} := \text{Normalize}_{\mathcal{A}}^{>}(\mathcal{C})$
    END
  END
END
RETURN $\text{trivial}(\mathcal{C})$

computer architecture rewards smaller process sizes with faster execution. More importantly, decent resource consumption is an essential prerequisite for getting provers productive in practice and for increasing acceptance by the users.

As an aside, any overhead for the construction of the detailed proof object is completely removed from the proof search (Sect. 3.3). It even becomes possible to depart from the given-clause algorithm. – This paper is a revised and extended version of [HL01]. We use standard notions and notations (cf. [DP01]).

2 An inspection of WALDMEISTER's proof procedure

2.1 The proof procedure itself

The procedure (cf. Alg. 1) is an instance of the *given-clause algorithm* [McC97, Chap. 5.5.1]. The input consists of an equation set $\mathcal{E}$, the *axioms,* a ground equation set $\mathcal{C}$, the *conjectures,* a reduction ordering $>$ total on ground terms, and a *weighting function* φ mapping equations to *weights.*

The saturation is performed in a cycle working on a set $\mathcal{A}$ of *active facts* that participate in inferences and a set $\mathcal{P}$ of *passive facts* waiting to become members of $\mathcal{A}$. The function φ heuristically assesses passive facts. It has to ensure that every passive fact becomes minimal at some point in time and hence selected. This guarantees the fairness of the proof procedure. We require φ to induce a total ordering on the equations that have been assessed so far. Typical examples such as $\varphi(s{=}t) = |s| + |t|$, i. e. the sum of term lengths, only induce quasi-orderings. These can be refined lexicographically with some timestamp, or with parents numbers and overlap position. Weights therefore are tuples with several components. Within limited storage, each of these components is bound, and so is the space needed to store it. In that sense, weights need constant space.

The active facts $\mathcal{A}$ induce, via their orientable instances $\mathcal{A}^{>}$, an ordered rewrite relation $\rightarrow_{\mathcal{A}}$ and thereby a normal form relation $\rightarrow^{!}_{\mathcal{A}}$. We stipulate a total *normal form function* $\mathrm{Normalize}^{>}_{\mathcal{A}} \subseteq \rightarrow^{!}_{\mathcal{A}}$ that is deterministic for any parameter value of $\mathcal{A}$ and $>$. Furthermore, an *interreduction function* $\mathrm{Interred}^{>}$ is needed that, given $\mathcal{A}$ and a new equation e, returns both those active facts that are e-reducible and the remainder of $\mathcal{A}$. Finally, let $\mathrm{CP}^{>}(e, \mathcal{A})$ denote the set of all critical pairs between the equation e and all equations in $\mathcal{A}$.

We comment now on some details of the proof procedure (cf. Alg. 1). (L. 2) The saturation proceeds as long as (i) the conjectures have not become trivial, i. e. equations of the form $t{=}t$, and (ii) there are still passive facts. (L. 3) Select an equation e from $\mathcal{P}$ which has minimal weight with respect to φ. (L. 4) Skip if a parent equation has been reduced meanwhile. The critical pair has not participated in any inference and hence is redundant. (L. 6) Skip if e has become trivial now, or redundant in some other way, e. g. via subsumption. (L. 10) Normalize the generated critical pairs and the reduced active facts, and add the non-redundant ones to $\mathcal{P}$.

For this proof procedure, the following two invariants hold: (i) Every non-redundant one-step conclusion of $\mathcal{A}$ is contained in $\mathcal{A} \cup \mathcal{P}$. (ii) The active facts $\mathcal{A}$ are completely interreduced. Hence, in case of termination with $\mathcal{P} = \varnothing$, $\mathcal{A}$ is an interreduced rewrite system, convergent on ground terms and equivalent to $\mathcal{E}$, i. e. a decision procedure for the uniform word problem of $\mathcal{E}$. Furthermore, if φ ensures fairness, then the procedure demonstrates any valid identity on ground terms in $\mathcal{C}$ within finite time.

In the following sections we will denote by $\mathcal{A}_i$ the value of $\mathcal{A}$ after the i-th iteration of the loop core in lines 7–11; this iteration count gives an abstract notion of time. The sets $\mathcal{A}_i$ induce the rewrite relations $\rightarrow_i \; := \; \rightarrow_{\mathcal{A}_i}$ and the normal form functions $\mathrm{Norm}_i := \mathrm{Normalize}^{>}_{\mathcal{A}_i}$.

The passive facts are, as can be seen in the proof procedure, subject to normalization only right after their generation and in case they get selected. This variant of given-clause algorithm is called *Discount loop*. In contrast, within an *Otter loop* the whole set of passive facts is normalized in every iteration of the cycle. Line 10 then would read "$\mathcal{P} := \mathrm{Normalize}^{>}_{\mathcal{A}}(\mathcal{P} \cup P_1 \cup P_2)$" instead. Some Otter loop implementations moreover use the rewrite relation generated by the union of active and passive facts, i. e. they always employ $\mathrm{Normalize}^{>}_{\mathcal{A} \cup \mathcal{P}}$ instead of $\mathrm{Normalize}^{>}_{\mathcal{A}}$.[1] The loop variants are named after well-known respective implementations, cf. [ADF95] and [McC94]. An early experimental comparison of both loops was done in [Küc82, p. 131], where, based on observations when completing the group axioms using the Aldes Data-Type Completion System, preference is given to the Otter loop. The loop names have been coined in [RV00, Chap. 2 and 6], where also extensive experiments of both variants are reported. A discussion within the frame of resolution and superposition is contained in [Wei01, Sect. 3 and 5.3].

[1] More precisely, this involves a more subtle form of interreduction. Note also that then the orphan predicate (L. 4) requires tagging passive facts that have ever participated in reductions, and that only untagged orphans may be deleted.

It is clear that, since nothing is done with passive facts but selection, the DISCOUNT loop leads to a simpler system design, because no term indexing is necessary on the elements of $\mathcal{P}$ as would be the case if $\mathcal{P}$ were to be kept fully normalized. Apart from relieving the system developers, this should also reduce the system run time, presuming that the search behavior of the prover is essentially the same, which admittedly is an issue still under discussion.

Interesting related work on the resource problem in theorem proving can be found in [RV00]. The setting there is to attack a proof problem *within some given time limit*. A *limited resource strategy*, LRS for short, is developed that explicitly takes this limitation into account in order to analyze a larger part of the search space in the time given. To this end, the prover keeps track of the average time spent for processing a selected passive fact. This allows to estimate the number N of facts that still can be activated before the time limit is reached. Hence $|\mathcal{P}| - N$ passive facts are not likely to be selected; and so the $|\mathcal{P}| - N$ heaviest ones of them are discarded. LRS is a refinement of the weight-limit strategy of OTTER where clauses are discarded if their weight exceeds a given, fixed bound. For the VAMPIRE system the authors observe an increase in performance with LRS. The approach pays off for systems implementing the OTTER loop, since fewer passive facts have to be kept and indexed. As to the DISCOUNT loop, space savings are to be expected there, but no speed-ups, since passive facts are not indexed. – The setting we are dealing with in this paper is a different one, namely running a prover without time limit; and the specialized compression techniques developed here will result in more memory savings.

2.2 Aspects of space consumption

Typically $\mathcal{P}$ contains far more equations than $\mathcal{A}$, as a rule of thumb $|\mathcal{P}| = O(|\mathcal{A}|^2)$ on the average (i. e. one can fit parabolas into the plot of these quantities). Therefore the space requirements of WALDMEISTER are dominated by the representation of the passive facts. Hence, methods to save space mainly have to tackle $\mathcal{P}$. Up to now, we have used simple *compression schemes* to minimize the space needed for single elements. Within WALDMEISTER, this can be done in three ways: (i) Not at all, i. e. flatterms are employed at the cost of 12 bytes per symbol. (ii) Stringterms: Terms are stored simply as strings of symbols, at the cost of one byte per symbol. (iii) Overlap representation: After heuristical assessment, the terms themselves are thrown away, and just minimal reconstruction information is kept, at constant cost.

The influence of these representations on the process size over abstract time can be studied in Fig. 1. For each variant, the WALDMEISTER system was run on the ROB001-1 proof task from the TPTP problem library [SS99] using an UltraSPARC-IIi workstation. The process size minus the space consumption of $\mathcal{P}$ serves as a baseline (lowest curve). Stringterms allow compression rates of about 10, which is comparable to what can be achieved with a shared term representation [LS01]. The overlap representation improves on this by another factor of nearly 10. But still here the overall space consumption, exceeding 1 GByte

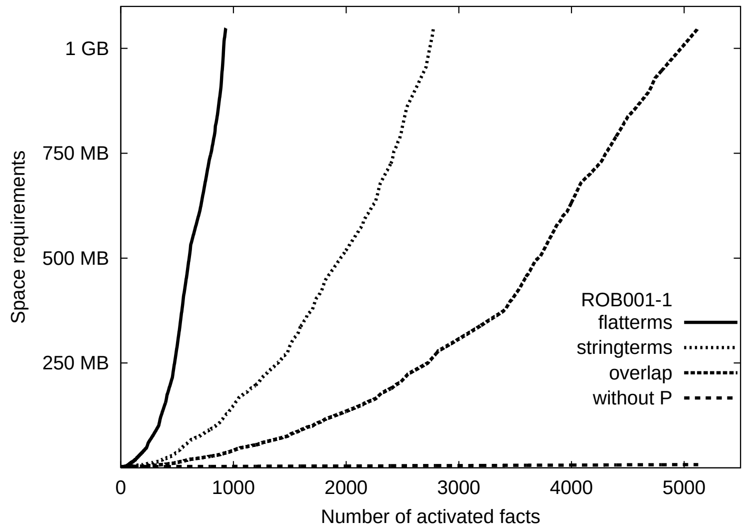

Fig. 1. Size of the process over abstract time

after some thousand activation steps, is mostly due to the data structure for $\mathcal{P}$, which mirrors the estimated quadratic relation between $|\mathcal{A}|$ and $|\mathcal{P}|$.

The above-mentioned overlap representation is apparently the best one in terms of space. But – contrary to one's expectations – it may influence the behavior of the proof search. To quantify the effect, we performed the following experiment: On the 431 unit equality proof problems in the current TPTP release [SS99] WALDMEISTER is run with six different settings, two orderings (KBO/LPO) combined with three general purpose weighting functions (addweight/mixweight/gtweight). For each of the 2586 cases we compare the prover's behavior using stringterms with that using overlap representation.

Within the resource limit of 120 seconds on a 1 GHz Pentium III system 1819 cases are solved by both versions and 699 by none. Whereas in 68 cases the run with the stringterm representation succeeds, and that with the overlap representation does not, there is no case where the opposite happens. From the 1819 cases solved by both versions, 1332 ($\approx$ 74 %) show no difference. In 385 ($\approx$ 21 %) cases, the stringterm version is better, in 102 ($\approx$ 6 %) the overlap version. Comparing the degree of difference, the improvements of the overlap representation are rather modest. In the other direction, however, the effects are stronger, e. g. there are numerous examples which are solved in under 5 seconds by the stringterm version and encounter a time-out with the overlap version.

On the inference level, the reason for the differences is the following: When a critical pair e is generated at time i, it is normalized to $\mathrm{Norm}_i(e)$ and inserted into $\mathcal{P}$. Let e be selected at time $j \gg i$. The reconstruction process needed by overlap representation then yields $\mathrm{Norm}_j(e)$, whereas with stringterms the critical pair would enter the completion loop in the shape $\mathrm{Norm}_j(\mathrm{Norm}_i(e))$. Since confluence is missing during completion, these shapes may differ.

With a detailed analysis of individual proof runs we have identified two main causes for the changes in the search behavior. (i) One version activates an equation that is shown to be redundant by the other. This incurs the generation of additional redundant critical pairs, so that the first version performs worse. (ii) Occurring less often, the size of an activated equation recomputed from overlap representation is larger. Since the weighting function φ typically prefers small equations, this disturbs the search. Furthermore, larger equations are less useful for simplification. Here the stringterm version behaves better.

These two phenomena are are consequence of the differences between Norm_i and Norm_j. Analyzing single cases on the level of single rewrite steps, we found e. g. that newer rewrite rules, with larger right-hand sides, were favored over the ones originally used. Or, a rule was no longer available due to interreduction. Since the detailed analysis is very costly and the cases investigated show no dominant pattern, we cannot give conclusive explanations.

3 The new approach

To reduce the memory consumption, which is mostly due to the subsystem $\mathcal{P}$, we develop further the idea of compression and reconstruction at need. The key idea of our approach is to represent *whole subsets* of $\mathcal{P}$ by *single entries*. These subsets are not arbitrary, but reflect the structure of the given-clause algorithm. The first kind of sets contains the critical pairs belonging to two *p*arent equations e_i and e_j. We therefore call them *p-sets*. The second kind of sets, called *a-sets*, consists of all the critical pairs jointly generated at the *a*ctivation of one given clause. Since these sets are structured they can be represented in the same compact way as individual entries. Especially, we will show a variant that guarantees that the size of the representation of $\mathcal{P}$ is linear in abstract time.

For intrinsic completeness reasons of our approach that will be worked out in Sect. 3.1, an important prerequisite is the availability of the *collected history of* $\mathcal{A}$, i. e. all the preceding states $\mathcal{A}_1, \mathcal{A}_2, \ldots$ of the active facts, to re-induce the corresponding normalization functions Norm_i. An elegant and efficient realization will be discussed in Sect. 3.2.

3.1 The new design of the $\mathcal{P}$ subsystem

The task of the subsystem $\mathcal{P}$ is to fairly enumerate all inserted equations with the help of a weighting function φ. $\mathcal{P}$ communicates with its environment via two methods: "insert" inserts all (non-redundant) critical pairs between an equation e_i and $\mathcal{A}_i$ at once. Additionally, all equations interreduced by e_i are inserted. The other method is "deleteMin". It extracts the minimal equation from $\mathcal{P}$. Concerning the abstractly formulated Alg. 1, deleteMin realizes the functionality for line 3, and insert is needed in line 10.

We require an implementation of $\mathcal{P}$ to be neutral with respect to inferences and proof search. Even if it performs internally some inferences for recomputations it shall appear from the outside like a mere container, namely as a priority queue storing the unmodified equations augmented with their weight computed by φ. Of course, this idealized *reference implementation* assumes the availability of unlimited memory. An actual implementation shall then show the same input/output (I/O)-behavior. That is, for the same sequence of insert and $\mathsf{deleteMin}$ operations (with the same arguments for insert), it shall return the same equations for $\mathsf{deleteMin}$ operations as the reference implementation.

In a first step, we only consider the use of compact representations for individual elements. Given two equations e_i and e_j, $i \geq j$, the critical pair at the overlap position p is uniquely determined. The position information can be encoded in a single integer, e. g. as number of the top symbol of the subterm identified by p when the symbols are counted from left to right. We extend p to an *extended position* xp with 3 bits. These encode the sides that are overlapped and whether p is a position of e_i or e_j. If $s{=}t$ is the critical pair, its weight w is calculated as $w = \varphi(\mathrm{Norm}_i(s{=}t))$. It can be represented within constant space via its *overlap entry* $\langle w, i, j, xp\rangle$. Equations stemming from interreduction can be encoded in a similar way: When e_i gets reduced by e_j, $j > i$, the entry is $\langle w, i, j, \dagger\rangle$, with $w = \varphi(\mathrm{Norm}_j(e_i))$ and a special symbol $\dagger$.

Since the data in the entries is sufficient for the reconstruction of the equations the following lemma is an easy consequence of the availability of the different Norm_i:

Lemma 1. *An implementation of $\mathcal{P}$ using only overlap and interreduction entries will show the same I/O-behavior as the reference implementation.* $\square$

In the second step, we define entries that represent more than one equation. *p-entries* arise from individual entries by abstracting from the extended position component and replacing it by a wildcard $*$. More formally the p-entry $\langle w, i, j, *\rangle$ represents all individual entries $\langle w', i, j, xp'\rangle$ with $w' \geq w$. Note that p-entries are a refinement of p-sets: they do not represent *all* critical pairs between the parents, but only the heavier ones. This allows to keep the "better" critical pairs individually. Abstracting again, now from the second equation, we can represent in an *a-entry* $\langle w, i, *, *\rangle$ all critical pairs $\langle w', i, j', xp'\rangle$ with $w' \geq w$ that are computed at the activation of e_i, hence $j' \leq i$. For the same reasons as above we distinguish between a-entries and a-sets analogously to p-entries and p-sets.

These new entry types allow the realization of stronger forms of compression. As building blocks for concrete algorithms we define the following conversions:

p-compression: $\mathcal{P} \cup \{\langle w', i, j, xp\rangle \mid w' \geq w\} \rightsquigarrow \mathcal{P} \cup \{\langle w, i, j, *\rangle\}$ if there is neither some p-entry for e_i and e_j nor an a-entry for e_i in $\mathcal{P}$.

a-compression: $\mathcal{P} \cup \{\langle w', i, j, xp\rangle \mid w' \geq w\} \rightsquigarrow \mathcal{P} \cup \{\langle w, i, *, *\rangle\}$ if there is neither some p-entry for e_i and some e_j nor an a-entry for e_i in $\mathcal{P}$.

p-decompression: $\mathcal{P} \cup \{\langle w, i, j, *\rangle\} \rightsquigarrow \mathcal{P} \cup \{\langle w', i, j, xp\rangle \mid$ there exists a critical pair $s{=}t$ between e_i and e_j at the extended position xp, $s'{=}t' = \mathrm{Norm}_i(s{=}t)$, $s' \not\equiv t'$, $w' = \varphi(s'{=}t')$, $w' \geq w\}$

a-decompression: $\mathcal{P} \cup \{\langle w, i, *, * \rangle\} \rightsquigarrow \mathcal{P} \cup \{\langle w', i, j, xp \rangle \mid e_j \in \mathcal{A}_i$, there exists a critical pair $s{=}t$ between e_i and e_j at the extended position xp, $s'{=}t' = \mathrm{Norm}_i(s{=}t)$, $s' \not\equiv t'$, $w' = \varphi(s'{=}t')$, $w' \geq w\}$

The side conditions imply that the intersection of two a- or p-entries is empty, which gives the important invariant that a critical pair is represented exactly once, either in an individual overlap entry or in an a- or p-entry.

Lemma 2. *p-decompression is the exact inverse of p-compression, a-decompression is the exact inverse of a-compression. The conversions preserve the property that any critical pair is represented once.* □

This lemma relies on the availability of the different Norm_i: The compression steps are parameterized by a weight limit w. Consider critical pairs descending from a pair e_i, e_j of parents (p-compression) respectively from a single parent e_i and $\mathcal{A}_i$ (a-compression). These have a weight which has been determined with respect to their normal form at generation time. We want to group only those critical pairs together that have a sufficiently heavy weight, since the smaller ones have already been selected. The problem arises in the corresponding decompression step. For every overlap between e_i and e_j respectively $\mathcal{A}_i$, we have to (i) rebuild the critical pair, (ii) normalize it, (iii) compute its weight, (iv) compare that weight to the w parameter of the group entry, and (v) add the normalized critical pair to the decompressed set in case the weight is not too small. If in (ii) we employ the normalization function of generation time, then we get the original normal form back and hence in (iii) the original weight, so that the decision in (iv) gives the correct result. However, with the current normalization function in (ii), we might obtain a different normal form, possibly with smaller weight. If that weight were smaller than w, then the critical pair erroneously would not be included in the decompressed set and hence not be considered in the completion process. – Without the previous Norm_i available, it would not be possible to discard the light-weighted critical pairs. They hence would, unless proven redundant by the current normalization, re-enter the saturation process. So the implementation of $\mathcal{P}$ would influence the behavior of the proof search, in addition to the weighting function φ. This influence will have to be studied when an implementation is at hand.

When the deleteMin operation is slightly modified, Lemma 1 carries over: Whenever a p- or a-entry becomes minimal the appropriate decompression conversion is called before deleting the minimal entry.

In a third step, we will now sketch a realization of $\mathcal{P}$ with a space usage which is linear with abstract time.

The idea is to store the light-weighted critical pairs individually in a constant-size buffer B and to compress only the heavier-weighted ones into appropriate a-sets in a buffer B' that grows linearly with abstract time. For each e_i it has either an a-entry or an interreduction entry[2]. The intention is to use B as some

[2] This requires to perform the orphan criterion actively, that is, as soon as an element of $\mathcal{A}$ is reduced, all accompanying orphans in $\mathcal{P}$ are deleted.

kind of *cache* for critical pairs with small weight. All the remaining critical pairs are represented in B'. This design reduces the need of recomputation. As a consequence of the previous lemmas we get the desired result:

Corollary 1. *With the accumulated history of $\mathcal{A}$ available, it is possible to implement a $\mathcal{P}$-subsystem which has a space usage which is linear with abstract time and which shows the same I/O-behavior as the reference implementation.*

3.2 The set of active facts $\mathcal{A}$ and its accumulated history

The problem we want to tackle now is how to efficiently provide the normal form function Norm_i at time $j \geq i$. Usually the normal form of a term t is computed by traversing its subterms in some fixed order (*reduction strategy*) up to the first one that is an instance of the left-hand side of some active fact. If such a subterm is not found, t is irreducible; otherwise the subterm is replaced with the instance of the right-hand side, and the traversal starts all over again.

For such normal form functions, remembering Norm_i reduces to remembering the generalization retrieval with respect to $\mathcal{A}_i$. In today's provers, this retrieval is based on some kind of indexing technique; see [Gra96] and [RSV01] for an overview. It is not necessarily unique, since more than one left-hand side may fit. For performance reasons *first-fit retrieval* is commonplace. We therefore need to remember the corresponding generalization functions Match_i.

In the WALDMEISTER system, *perfect discrimination trees* are employed to represent the active facts. That is a trie-like structure where terms are interpreted as sequences of symbols in left-to-right traversal, and every edge carries a symbol as a label. The construction principle is that, for every subtree, its term entries all have the same prefix, namely the sequence of labels on the path from the root to that very subtree. To increase the amount of this sharing of prefixes, variables of indexed terms are *normalized*, i. e. renamed to $x_1, x_2, \ldots$ in order of their left-to-right fresh appearance.

The retrieval algorithm then works in a backtracking fashion (cf. [Hil00]), on a triple of data: a pointer into the query term which is traversed in left-to-right order; a reference to the subtree to be searched through; and a partial substitution constructed so far. For every triple, there may be up to three possibilities to proceed: (i) The subterm starts with a function symbol, and the subtree has a descendant via an edge labeled with that very symbol. (ii) There is a descendant corresponding to a yet unbound variable. (iii) The same but with a variable already bound, such that its substitute is a prefix of the current query part. We stipulate that alternative possibilities are tested in that arrangement, i. e. first function symbols, and then variables by decreasing index. The algorithm succeeds in case it reaches a leaf node.

Consider as an example the search for generalizations of $f(a, a, b)$ in the index for $\mathcal{A}_3$ depicted in Fig. 2 (b); that index contains the entries $f(a, b, a)$, $f(x_1, x_2, b)$, and $f(x_1, x_1, x_2)$. The retrieval starts with the empty substitution in node 1. There is a descendant via the top function symbol f; so 2 is reached with the query rest $a\,a\,b$. This node has two successors that might match: one

(a)

1 Insert $f(a, b, a)$
2 Insert $f(x_1, x_2, b)$
3 Insert $f(x_1, x_1, x_2)$
4 Insert $f(a, x_1, x_2)$
Delete $f(a, b, a)$

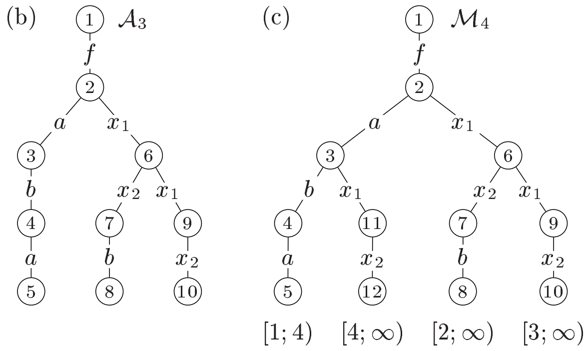

Fig. 2. (a) Incremental construction of $\mathcal{A}_1 \ldots \mathcal{A}_4$ (b) and (c) Indexes for $\mathcal{A}_3$ and $\mathcal{M}_4$

via the constant symbol a, and another one via the variable x_1. According to the above convention, 3 is visited first, where the query rest is $a\,b$. There is no suitable successor, so backtracking is invoked to 2 where the descent to 6 via the edge labeled x_1 is done, establishing a binding $\{x_1 \mapsto a\}$. The query rest is $a\,b$, and the node offers two possibilities to continue, namely with the free variable x_2 and the already bound one x_1. By convention x_2 is checked first, with binding $\{x_2 \mapsto a\}$, and thereby reaching 7 with query rest b. This is followed by the descent to 8 whereby a leaf node is reached, implying that the entry $f(x, y, b)$ matches $f(a, a, b)$ with the substitution $\{x \mapsto a, y \mapsto b\}$. Note that also the leaf node 10 corresponds to a successful match, namely $f(x, x, y)$ with $\{x \mapsto a, y \mapsto b\}$, but it is not the first one found.

The tree traversal induces an ordering on prefix symbol sequences in the index. This can be formalized with an appropriate total precedence $\sqsupset$ on the function symbols $\mathcal{F}$, extended to $\mathcal{F} \cup \mathcal{X}$ via $f \sqsupset x_j$ and $x_i \sqsupset x_j$ iff $i >_{\mathbb{N}} j$. Then the lexicographic extension $\sqsupset^*$ is a total ordering on symbol sequences, and also on terms considered as sequences of symbols. We then can formulate the following invariants for the retrieval process reaching a tree node:

1. The sequence of query term symbols considered so far equals the sequence of labels from the root to the current node under the substitution at hand; and hence every term indexed by the current subtree also has that prefix.
2. Consider a node which has a label sequence greater with respect to $\sqsupset^*$ than the current one. Then the corresponding subtree contains a match only if it contains the current node.

For the retrieval example at hand when e. g. visiting node 6, this means first that the entries $f(x_1, x_2, b)$ and $f(x_1, x_1, x_2)$ under the substitution $\{x_1 \mapsto a\}$ start with $f\,a$, and second that the nodes 3, 4 and 5 do not lead to successful matches, but 1 and 2 might. – These invariants guarantee not only that generalization retrieval is sound and complete, but also that the match found, if at all, is the greatest one with respect to $\sqsupset^*$. That is, the algorithm without any overhead returns a uniquely determined match. We will exploit this property for efficiently remembering Match_i.

Corresponding to the state sequence of the active facts $\mathcal{A} = \mathcal{A}_1, \mathcal{A}_2, \ldots$, we construct a *memory sequence* $\mathcal{M} = \mathcal{M}_1, \mathcal{M}_2, \ldots$ containing all the active facts that have ever been active so far, along with an additional activation and deactivation timestamp. The key point now will be that a single additional discrimination tree is sufficient to represent $\mathcal{M}$.

An insertion into the index for $\mathcal{A}$ is reflected by a parallel insertion into the index for $\mathcal{M}$, with an appropriate activation timestamp attached to the element. A deletion from the index for $\mathcal{A}$, however, is not mirrored by a deletion from the index for $\mathcal{M}$, but just by setting a deactivation time for the element. We need to define a generalization function $\mathrm{Match}_{i,j}$ that, if called at a current time $j \geq i$, uses the index for $\mathcal{M}_j$ to reconstruct the operation Match_i once performed on the index for $\mathcal{A}_i$.

For doing so, we only have to impose an additional, simple condition in the above standard operation for the retrieval of generalizations: When a leaf node is reached, this implies success only if the attached activation and deactivation timestamps comply with i.

Figure 2 shows a sequence of activation and deactivation steps that results in an index for $\mathcal{M}_4$ to memorize the sequence of active fact sets $\mathcal{A}_1, \ldots, \mathcal{A}_4$. Above we have shown how a generalization for $f(a, a, b)$ is found in the index for $\mathcal{A}_3$. This can now be reconstructed with the index for $\mathcal{M}_4$: The retrieval operation successively visits the nodes 1 and 2 just like in the index for $\mathcal{A}_3$. In node 3, there now is an appropriate successor via x_1 that leads to 11 and the leaf node 12. That node holds the matching entry $f(a, x_1, x_2)$, but its timestamps indicate that it is not in $\mathcal{A}_3$. Therefore, backtracking to 2 is invoked, and thereafter the nodes 6, 7 and 8 are inspected. That leaf node holds an entry with appropriate timestamps, and is hence returned. In fact, it is the one previously found in the index for $\mathcal{A}_3$.

The reason behind this can be seen in the refinement of the second invariant for the adapted retrieval operation: Any node with a label sequence greater than that of the current one leads to a *timely* match only if it leads to the current node. Therefore this retrieval returns the $\sqsupset^*$-greatest timely match in $\mathcal{M}_j$. By construction of the $\mathcal{M}$ sequence, the set of all timely matches is equal to the set of all matches in $\mathcal{A}_i$. The uniqueness of the greatest match now implies that both operations return the same; so we have $\mathrm{Match}_{i,j} = \mathrm{Match}_i$. – Note that the timestamp constraints could also be attached not only to leaves, but also to inner nodes, as union or covering of the timestamps of the descendants. In our example backtracking then would have been invoked not in node 12, but already in 11.

Let us finally remark that here a crucial prerequisite for efficiently remembering the generalization functions Match_i has been that the sets of all matches are naturally ordered such that the retrieval automatically stops with the unique greatest one. It is unclear to us how the same result can be achieved with other indexing schemes, especially if the index for a term set is not unique.

3.3 The Waldmeister loop

All in all, the elaborate scheme of compression and decompression results in a proof procedure that considerably departs from the original Discount loop. From a more abstract point of view, the differences can be interpreted as follows: The generation the critical pairs is delegated from the main loop to $\mathcal{P}$. It is then the task of the subsystem $\mathcal{P}$ to enumerate all non-redundant critical pairs in a fair way. $\mathcal{P}$ is seen as a black box here. It may perform the same normalizations as the main loop did, but these are no "real" inferences, they are for heuristical purposes only. Conceptually, the "real" inferences occur merely at the point of selection; only "good" critical pairs ever enter the completion loop.

This opens the avenue for further developments: Either one can replace the initial normalizations by some approximations. Or, one can completely depart from the given-clause algorithm and simulate e. g. the *pair algorithm* [McC97]. In this alternative a pair of activated equations e_i and e_j is selected via some heuristic ψ and then all critical pairs between them are generated. After simplification all the non-redundant ones are activated and the next pair is determined. In our framework, $\mathcal{P}$ essentially consists of p-entries of the form $\langle \psi(e_i, e_j), i, j, * \rangle$. When a p-decompression occurs, we set the weight of the individual entries to the smallest possible value to ensure that all individual entries are handled before the next p-entry is considered. If for an individual entry the values of ψ on the parents and φ on the equation are combined into one, we get hybrid versions.

Of course, the whole approach is suitable for reasoning with full clauses as well. The only requirement is that any sequence of simplification steps and redundancy tests performed on a passive fact at generation time can *identically* be reproduced at selection time. For binary resolution, the entry format simply has to be adapted such that the position information encodes the literal resolved upon. The extension to superposition or hyperresolution is an easy exercise as well. The more complex the logical objects dealt with are, the more the benefits of the approach, take for example clauses augmented with constraints.

As a final point, since the collected history of $\mathcal{A}$ is available, it is for the generation of a proof object no longer necessary to protocol the inferences of the proof search. When a proof has been found, the contributing active facts are determined. These are then reconstructed while full logging of the inferences is enabled, delivering a detailed proof object. This separates proof search and proof object construction, a further win in architectural clarity.

4 First practical results

Several important questions remain: What is the trade-off between space savings and recomputation effort in practice? What are typical values for the average sizes of p-sets and a-sets? Is it worthwhile to modify core parts of existing systems for the integration of the new technique? As it turns out we can answer these question by retrofitting Waldmeister to write run-time traces logging the equations inserted into or deleted from $\mathcal{P}$. These traces allow to gather the

Table 1. Characteristic data of representative examples

problem	activated facts	critical pairs generated	critical pairs inserted	critical pairs deleted	avg. size of p-sets	avg. size of a-sets
GRP164-1	1 089	1 460 000	963 000	80 900	1.40	486
X6	1 408	2 760 000	2 650 000	47 200	3.64	1 960
RNG027-5	1 089	3 270 000	3 170 000	310 000	6.46	3 000
LAT038-1	2 753	3 760 000	1 630 000	179 000	1.68	1 370
ROB006-1	5 588	10 300 000	10 100 000	82 800	2.45	1 850
ROB001-1	6 220	42 400 000	42 400 000	37 800	8.07	6 820

desired data. Furthermore, we can then simulate different compression schemes and can thus estimate the benefits of an implementation.

4.1 Writing and analyzing $\mathcal{P}$-traces

Waldmeister is modified to write the following information into a file: for each generated critical pair the contributing parents, a unique identifier, and whether it can be shown to be redundant or if it is inserted into $\mathcal{P}$. For each element selected from $\mathcal{P}$ the same identifier is written again. Each deletion of an element of $\mathcal{A}$ is protocoled as well to allow a proper handling of interreductions and orphans in the simulations.

The proof problems we consider are mostly taken from different domains of the TPTP library [SS99] to cover a wide range, except for problem X6 that each ring where $x^6 = x$ is commutative. For all these problems we choose parameter settings for Waldmeister such that they can be proved within several minutes. As a contrast, we add the problem ROB001-1 which suffers a memory overflow even with the overlap term representation (cf. Sect. 2.2 and Fig. 1).

In Table 1 we summarize relevant data extracted from the traces. Despite that all the chosen examples generate more than a million critical pairs, the average size of p-sets is rather small, with a rather homogeneous distribution. We conclude that the use of p-entries for compression purposes is rather limited. Their real potential lies in the emulation and variation of the pair algorithm. The distribution of the sizes of the a-sets is far more widespread. After a short amount of abstract time we can expect them to be at least in the range of several hundreds of equations. So their use opens the real opportunity for the drastic reduction of the space requirements.

4.2 Simulations

The complete implementation of the developed scheme requires a significant amount of developer time since core modules of a theorem prover are affected. Fortunately, it is possible to simulate the component $\mathcal{P}$ in such a degree that we can estimate the trade-offs between space usage and time consumption.

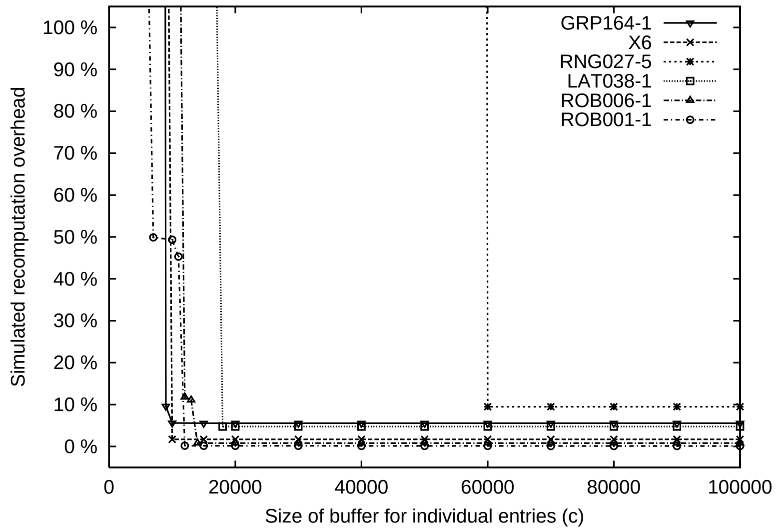

Fig. 3. Recomputation effort as function of size parameter c, for examples of Table 1

We determine the amount of computation effort as the number of critical pairs which have to be computed and normalized. Thus, the costs for additional recomputations in case of a deleteMin operation or an decompression conversion is based on the number of affected equations. We choose this notion since the costs for each critical pair are in our experience quite evenly distributed. To get a useful measure across the different examples we set the computation effort of the simulated strategy in relation to that of the reference implementation.

From our simulations, we present the following strategy for the two-buffer approach, which shows a quite good behavior. The parameter c determines the size of buffer B for the individual entries. Buffer B' for the a-entries grows with abstract time. Furthermore, we have a *weight limit* w_B which is initially ∞. For each newly inserted clause the weight is compared with w_B. If it is smaller the overlap representation is inserted into B. Otherwise the corresponding a-entry in B' is modified and the equation is discarded. When B gets full, the heavier half of entries is discarded (the a-entries in B' are updated accordingly). Then w_B is determined as the weight of the maximal clause in B. For the selection of clauses from $\mathcal{P}$ the entry with the minimal weight within B and B' is determined. If it is in B, the critical pair is reconstructed from the overlap taken out of B. In the other case, the a-entry is expanded.

Figure 3 shows the recomputation effort as a function of the size parameter c for the examples listed in Table 1. As can be seen this straight-forward strategy performs quite well. Even with a rather small value for c of say 20 000 all but one of the examples can be handled within a small overhead of less than ten per cent. To be useful in general, the strategy has to be extended to dynamically monitor the reconstruction overhead and react accordingly.

Comparing Fig. 1 in Sect. 2.2 with the results of the simulation, we can conclude that we will gain a huge reduction of the space usage of WALDMEISTER: Instead of wasting 1 GByte of memory a new implementation will be satisfied by less than 10 MByte, an improvement of two orders of magnitude for the whole system, and three orders of magnitude considering $\mathcal{P}$ alone.

References

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: a system for distributed equational deduction. In J. Hsiang, ed., *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, volume 914 of *LNCS*, pp. 397–402. Springer-Verlag, 1995.

[DP01] N. Dershowitz and D. A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, volume I, chapter 9, pp. 535–610. Elsevier, 2001.

[Gra96] P. Graf. *Term Indexing*, volume 1053 of *LNAI*. Springer-Verlag, 1996.

[Hil00] Th. Hillenbrand. Schnelles Gleichheitsbeweisen: Vom Vervollständigungskalkül zum WALDMEISTER-System. Diplomarbeit, Universität Kaiserslautern, Fachbereich Informatik, 2000. Available via http://www.mpi-sb.mpg.de/~hillen/documents/SchnellesBeweisen.ps.gz.

[HL01] Th. Hillenbrand and B. Löchner. The next WALDMEISTER loop (extended abstract). In H. de Nivelle and S. Schulz, eds., *Proceedings of the Second International Workshop on Implementation of Logics*, Technical Report MPI-I-2001-2-006, pp. 13–21. Max-Planck-Institut für Informatik Saarbrücken, 2001.

[Küc82] W. Küchlin. An implementation and investigation of the Knuth-Bendix completion procedure. Interner Bericht 17/82, Universität Karlsruhe, 1982.

[LS01] B. Löchner and S. Schulz. An evaluation of shared rewriting. In H. de Nivelle and S. Schulz, eds., *Proceedings of the Second International Workshop on Implementation of Logics*, Technical Report MPI-I-2001-2-006, pp. 33–48. Max-Planck-Institut für Informatik Saarbrücken, 2001.

[McC94] W. McCune. OTTER 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.

[McC97] W. McCune. 33 basic test problems: a practical evaluation of some paramodulation strategies. In R. Veroff, ed., *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, chapter 5, pp. 71–114. MIT Press, 1997.

[RSV01] I. V. Ramakrishnan, R. Sekar, and A. Voronkov. Term indexing. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, volume II, chapter 26, pp. 1853–1964. Elsevier, 2001.

[RV00] A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. Technical Report CSPP-7, University of Manchester, 2000. Accepted for publication in *Journal of Symbolic Computation*.

[SS99] C. B. Suttner and G. Sutcliffe. The TPTP problem library (TPTP v2.2.0). Technical Report 99/02, James Cook University, Townsville, 1999.

[Vor01] A. Voronkov. Algorithms, datastructures, and other issues in efficient automated deduction (invited talk). In R. Goré, A. Leitsch, and T. Nipkow, eds., *Proceedings of the First International Joint Conference on Automated Reasoning*, volume 2083 of *LNAI*, pp. 13–28. Springer-Verlag, 2001.

[Wei01] Chr. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, eds., *Handbook of Automated Reasoning*, volume II, chapter 27, pp. 1965–2013. Elsevier, 2001.