Term Ordering Diagrams

Márton Hajdu[1(✉)], Robin Coutelier[1], Laura Kovács[1], and Andrei Voronkov[2,3]

[1] TU Wien, Vienna, Austria
marton.hajdu@tuwien.ac.at
[2] University of Manchester, Manchester, UK
[3] EasyChair, Manchester, UK

Abstract. The superposition calculus for reasoning in first-order logic with equality relies on simplification orderings on terms. Modern saturation provers use the Knuth-Bendix order (KBO) and the lexicographic path order (LPO) for discovering redundant clauses and inferences. Implementing term orderings is, however, challenging. While KBO comparisons can be performed in linear time and LPO checks in quadratic time, using the best-known algorithms for these orders is not enough. Indeed, our experiments show that for some examples, term ordering checks may use about 98% of the overall proving time. The reason for this is that some equalities that cannot be ordered can become ordered after applying a substitution (post-ordered), and we have to check for post-ordering repeatedly for the same equalities. In this paper, we show how to improve post-ordering checks by introducing a new data structure called *term ordering diagrams*, in short TODs, which creates an index for these checks. We achieve efficiency by lazy modifications of the index and by storing and reusing information from previously performed checks to speed up subsequent checks. Our experiments demonstrate the efficiency of TODs.

1 Introduction

Superposition-based theorem provers commonly use simplification orderings on terms to restrict their search space [17]. All top performing provers from the last CASC competitions [25] – Vampire [11], iProver [8], E [23], and Zipperposition [28] – use the Knuth-Bendix order (KBO) [7] and some also use the lexicographic path order (LPO) [6]. There is a linear-time KBO algorithm [15] and a quadratic-time LPO algorithm [18], so implementing ordering comparisons does not seem to be challenging.

We denote by $\succ$ the simplification order used by the superposition calculus, and we call an *ordering comparison* the operation of checking whether $s \succ t$ holds for two terms s and t. Although there are very efficient algorithms for ordering comparison, surprisingly, it can easily take up a significant part of a theorem prover's running time. For example, ordering comparison is required in the following frequently used inference rule called *demodulation*:

$$\frac{l \approx r \qquad C[l\sigma]}{C[r\sigma]} \quad \text{where} \quad \begin{array}{l}(1)\ l\sigma \succ r\sigma,\\ (2)\ C[l\sigma] \succ (l \simeq r)\sigma,\end{array}$$

between a unit clause consisting of an equality $l \approx r$ and a clause $C[l\sigma]$ containing a subterm $l\sigma$, where σ is a substitution. We can remove the right premise $C[l\sigma]$ after applying this rule, so demodulation is very valuable for keeping the search space smaller.

Post-ordering checks with unordered equalities. Demodulation is applied only when the side condition $l\sigma \succ r\sigma$ is satisfied. Based on properties of simplification orders, this side condition always holds when $l \succ r$. A more challenging case is when $l \not\succ r$ and $r \not\succ l$ (we say that $l \approx r$ is *unordered*), and we repeatedly apply demodulation with the same left premise $l \approx r$ and different right premises. We call the comparison of $l\sigma$ and $r\sigma$ a *post-ordering check* to emphasize that $l \approx r$ is not pre-ordered. The number of required post-ordering checks can be very large. For example, if we have a unit equality problem (which is not unusual in algebraic reasoning) and generate 10^6 clauses, the number of demodulation inferences can be of the order of 10^{12}. Even if only 1% of all equalities are unordered, we still can have 10^{10} post-ordering comparisons. Further, while KBO has a linear time algorithm [15], it still takes a significant time compared to other algorithms used by a theorem prover (such as matching or unification). Improving post-ordering checks could thus further improve equational reasoning.

Equality retrieval with post-ordering checks. This paper focuses on retrieving equalities that become ordered after applying a substitution. This is used in implementing the superposition and demodulation rules, after the retrieval of candidate equalities of the form $l \approx r$, for applying these rules with the same left-hand side l. This can be formulated as the following problem.

The Post-Ordering Problem

Given a finite set $\mathcal{E}$ of unordered equalities $l \approx r_1, \ldots, l \approx r_n$ with the same left-hand side and a substitution σ, *retrieve* from $\mathcal{E}$ equalities that satisfy $l\sigma \succ r_i\sigma$.

We can be interested in retrieving one, some, or all equalities. When this problem is used in demodulation, one normally first uses an index that retrieves a term l, and only then unordered equalities $l \approx r_1, \ldots, l \approx r_n$ that form the set $\mathcal{E}$. In this case, all equalities in $\mathcal{E}$ have the same left-hand side. We assume that the same set $\mathcal{E}$ of equalities can be repeatedly used for retrieval, interleaved with operations of adding new equalities to $\mathcal{E}$ or removing existing equalities from it. As such, the post-ordering problem becomes a *term indexing problem* [24], where the substitution σ is a *query substitution*.

Motivating Example. Let us illustrate the post-ordering problem using a KBO with a constant weight function. Let l and r_1 denote the terms $f(x, y)$ and $f(y, x)$, respectively, and let $l \approx r_1$ be an equality. Given a substitution σ, checking whether $l\sigma \succ r_1\sigma$ holds involves computing term weights and comparing symbol precedences of $f(x, y)\sigma$ and $f(y, x)\sigma$ recursively. Some of these operations are, however, independent of the substitution σ applied. For example, the weight of

$f(x,y)\sigma$ is the same as the weight of $f(y,x)\sigma$ independently of σ, so the weight check can be dropped. Simple analysis shows that checking $l\sigma \succ r_1\sigma$ can be simplified to an *equivalent* ordering check $x\sigma \succ y\sigma$. Note that this analysis has to be performed only once, so all consequent checks of $f(x,y)\sigma' \succ f(y,x)\sigma'$ for any substitution σ' can be immediately reduced to checking $x\sigma' \succ y\sigma'$.

We can do even better when we perform several consecutive checks. For example, suppose that we retrieve equalities in the same order and the next equality after $f(x,y) \approx f(y,x)$ is $f(x,y) \approx f(x,x)$. Then, if during the first check we established $y\sigma \succ x\sigma$, we can immediately conclude $f(x,y)\sigma \succ f(x,x)\sigma$.

Our contributions. This paper is based on ideas explained in the example above: (i) simplify ordering checks using the definition of KBO or LPO, and (ii) use the previous computation history to get rid of redundant checks. We bring the following contributions.

- We introduce a new data structure, called the *term ordering diagram (TOD)*, in Section 3.
- We describe TOD transformations in Section 4. These transformations use KBO/LPO properties and produce equivalent, yet more efficient TODs. [4]
- We propose an equality retrieval algorithm in Section 5 for solving the post-ordering problem using TODs. To the best of our knowledge, our work provides the first algorithm for efficiently solving post-ordering checks. It is also the first paper discussing runtime specialization of LPO checks, previously used for KBO ordering checks in [22] and earlier in [27,21] for other operations.
- We evaluate our work by implementing term ordering diagrams in Vampire and report on our results in Section 6. The use of TODs in forward demodulation, that is, demodulation applied to newly generated clauses from previously stored unit equalities [11], has significantly contributed to Vampire's success in the unit equality division (UEQ) of the CASC competition [25] in 2024.

2 Preliminaries

We work with a fixed signature $\mathcal{F}$ consisting of a finite set of *function symbols* with associated arities and consider an alphabet of *variables* $\mathcal{V}$; variables are not part of the signature. *Function symbols* are denoted by f, g, and variables by x, y, possibly with indices. *Terms* are defined in the standard way over $\mathcal{F} \cup \mathcal{V}$; variable-free terms are called *ground*. A *substitution* σ is a mapping from variables to terms, such that the set $\{x \mid \sigma(x) \neq x\}$ of variables is finite. We call a *simplification ordering* any ordering $\succ$ on terms that is

(1) *well-founded*: there is no infinite decreasing chain of terms $s_1 \succ s_2 \succ \ldots$,
(2) *stable under substitutions*: if $s \succ t$ then $s\sigma \succ t\sigma$,

[4] Detailed proofs can be found in the extended version of this paper [5].

(3) *monotonic*: for $f \in \mathcal{F}$ and terms $s_1, \ldots, s_n, s, t$ such that $s \succ t$, we have $f(s_1, \ldots, s_{i-1}, s, s_i, \ldots, s_n) \succ f(s_1, \ldots, s_{i-1}, t, s_i, \ldots, s_n)$,
(4) satisfies the *subterm property*: if t is a proper subterm of s, then $s \succ t$.

A *precedence relation*, denoted by $\gg$, is a total order on the signature $\mathcal{F}$. Given a precedence relation, a *weight function* is a function w from $\mathcal{F}$ to non-negative integers such that (i) $w(c) > 0$ for all constants c and (ii) if $w(f) = 0$ and f is unary, then f is the greatest element w.r.t. $\gg$. We will refer to $w(f)$ as the *weight* of f. We denote by w_0 the smallest weight of constants. For $p \in \mathcal{F} \cup \mathcal{V}$, we write $|t|_p$ to denote the *number of occurrences* of a variable or a symbol p in a term t. For example, $|f(x,x)|_f = 1$, $|f(x,x)|_x = 2$ and $f(x,x)_y = 0$. Let $\mathcal{P}(\mathcal{V})$ be the set of linear expressions over $\mathcal{V}$ with integer coefficients. The *weight of a term* t, denoted by $|t|$, is a linear expression in $\mathcal{P}(\mathcal{V})$ defined as:

$$|t| \stackrel{\text{def}}{=} \sum_{f \in \mathcal{F}} |t|_f \cdot w(f) + \sum_{x \in \mathcal{V}} |t|_x \cdot x$$

A substitution σ can also be considered as a mapping from linear expressions to linear expressions, as follows:

$$\sigma(\alpha_0 + \alpha_1 \cdot x_1 + \ldots + \alpha_n \cdot x_n) \stackrel{\text{def}}{=} \alpha_0 + \alpha_1 \cdot |x_1\sigma| + \ldots + \alpha_n \cdot |x_n\sigma|.$$

For example, if $w(f) = 2$, $w(a) = 1$ and $\sigma = \{x \mapsto a\}$, then $|f(x,x)| = 2 \cdot x + 2$ and $\sigma(|f(x,x)|) = \sigma(2 \cdot x + 2) = 2 \cdot |x\sigma| + 2 = 4$. It is not hard to argue that $|t\sigma| = \sigma(|t|)$. Let $e \in \mathcal{P}(\mathcal{V})$ be a linear expression. We call a substitution σ *grounding* for e if $\sigma(e)$ does not contain variables and $|x\sigma| \geq w_0$ for all $x \in \mathcal{V}$ such that $|x\sigma|$ is a constant. We write $e > 0$ if $\sigma(e) > 0$ for all grounding substitutions σ for e. We write $e \gtrsim 0$ if $\sigma(e) \geq 0$ for all grounding substitution σ for e.

The *Knuth-Bendix order (KBO)*, denoted by $\succ_{\mathsf{kbo}}$, is parameterized by a precedence relation $\gg$ and a weight function w. For terms s, t, we have $s \succ_{\mathsf{kbo}} t$ if:

(K1) $|s| - |t| > 0$, or
(K2) $|s| - |t| \gtrsim 0$, $s = f(s_1, ..., s_n)$, $t = g(t_1, ..., t_m)$ and $f \gg g$, or
(K3) $|s| - |t| \gtrsim 0$, $s = f(s_1, ..., s_n)$, $t = f(t_1, ..., t_n)$ and there exists $1 \leq i \leq n$ such that $s_i \succ_{\mathsf{kbo}} t_i$ and $s_j = t_j$ for all $1 \leq j < i$.

The *lexicographic path order (LPO)*, denoted by $\succ_{\mathsf{lpo}}$, is parameterized by a precedence relation $\gg$. Let s, t be terms with $s = f(s_1, ..., s_n)$. We write $s \succ_{\mathsf{lpo}} t$ if:

(L1) $t = f(t_1, ..., t_n)$ and there exists $1 \leq i \leq n$ s.t. $s_j = t_j$ for $1 \leq j < i$, $s_i \succ_{\mathsf{lpo}} t_i$, and $s \succ_{\mathsf{lpo}} t_k$ for $i < k \leq n$, or
(L2) $t = g(t_1, ..., t_m)$, $f \gg g$ and $s \succ_{\mathsf{lpo}} t_i$ for $1 \leq i \leq m$, or
(L3) $s_i \succeq_{\mathsf{lpo}} t$ for some $1 \leq i \leq n$.

It is known that every instance of LPO and KBO is a simplification order.

3 Term Ordering Diagram – TOD

To solve the post-ordering problem, we introduce the *term ordering diagram (TOD)* data structure (Definition 1). For retrieving equalities using TODs, we will need the following two operations on substitutions.

1. Given two terms s and t, compare $s\sigma$ and $t\sigma$ using $\succ$. We call the expression $s >^? t$ a *term comparison*. We consider a substitution σ as a mapping from term comparisons to the set of values $\{>, =, \not\geq\}$, as follows:

$$\sigma(s >^? t) \stackrel{\text{def}}{=} \begin{cases} >, & \text{if } s\sigma \succ t\sigma, \\ =, & \text{if } s\sigma = t\sigma, \\ \not\geq, & \text{otherwise.} \end{cases}$$

2. Given a linear expression e, check the sign of the linear expression $\sigma(e)$. We call the expression $e \geq^? 0$ a *positivity check*. We consider a substitution σ as a mapping from positivity checks to the set of values $\{>, \geq, \not\geq\}$, as follows:

$$\sigma(e \geq^? 0) \stackrel{\text{def}}{=} \begin{cases} >, & \text{if } \sigma(e) > 0, \\ \geq, & \text{if } \sigma(e) \gtrsim 0, \\ \not\geq, & \text{otherwise.} \end{cases}$$

   Note that positivity checks are only defined for KBO.

Before defining TODs, we illustrate how we solve the post-ordering problem.

Example 1. Let σ be a query substitution and $l \stackrel{\text{def}}{=} f(x, y)$, $r_1 \stackrel{\text{def}}{=} f(y, x)$ be terms. The rooted directed graph of Figure 1(a) illustrates the two key operations for retrieving the equality $l \approx r_1$ if $f(x, y)\sigma \succ f(y, x)\sigma$ holds. We first evaluate the *term comparison* $f(x, y) >^? f(y, x)$ in the top node. If $\sigma(f(x, y) >^? f(y, x))$ evaluates to $>$, the computation proceeds to the bottom node, which contains the equality $l \approx r_1$, which is then included in the result of the retrieval.

The evaluation of $\sigma(f(x, y) >^? f(y, x))$ in Figure 1(a) uses KBO, which in turn computes the linear expression $|f(x, y)| - |f(y, x)|$, performs a *positivity check* and proceeds with one of the subcases (K1)–(K3). The linear expression is 0 as both terms have the same number of f, x, and y symbols. Hence, the positivity check always results in $\geq$, which rules out subcase (K1). Similarly, (K2) is violated, as $f(x, y)$ and $f(y, x)$ both have f as top-most symbol.

Note that (K1) and (K2) are not applicable *regardless of the query substitution* σ, hence we can simplify $f(x, y)\sigma \succ f(y, x)\sigma$ into $(x \succ y \vee (x = y \wedge y \succ x))\sigma$. As $(x = y \wedge y \succ x)\sigma$ is false, this further simplifies into $(x \succ y)\sigma$. The term comparison $f(x, y) >^? f(y, x)$ of Figure 1(a) is thus turned into the (cheaper) term comparison $x >^? y$ shown in Figure 1(b).

Let us now consider an additional equality $l \approx r_2$, with $r_2 \stackrel{\text{def}}{=} f(x, x)$. Figure 1(c) displays the steps of retrieving $l \approx r_2$ if $f(x, y)\sigma \succ f(x, x)\sigma$ holds. Similarly to Figure 1(a), the *term comparison* $f(x, y) >^? f(x, x)$ is simplified using KBO properties into $(y - x > 0 \vee (y - x \gtrsim 0 \wedge y \succ x))\sigma$.

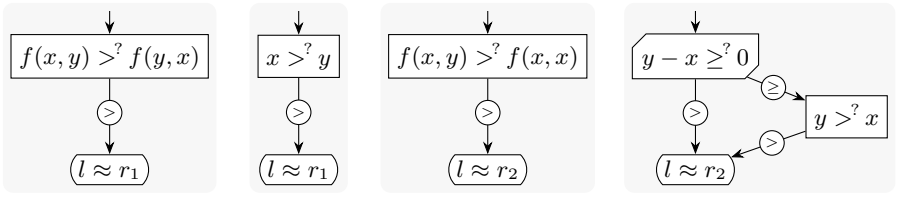


Fig. 1. Equality retrievals, where $l \stackrel{\text{def}}{=} f(x,y)$, $r_1 \stackrel{\text{def}}{=} f(y,x)$ and $r_2 \stackrel{\text{def}}{=} f(x,x)$. (a) and (b) show retrieval of $l \approx r_1$, where (a) contains a term comparison $l >^? r_1$ and (b) is a simplified version of (a) with term comparison $x >^? y$. Further, (c) and (d) show retrieval of $l \approx r_2$, where (c) contains term comparison $l >^? r_2$ and (d) is a simplified version of (c) with a positivity check $y - x \geq^? 0$ and term comparison $y >^? x$.

The simplified computation is depicted in Figure 1(d), where we first perform a *positivity check* $y - x \geq^? 0$, corresponding to deciding which of $\sigma(y-x) > 0$ or $\sigma(y-x) \gtrsim 0$ hold in the above formula, then performing at most one more term comparison. The computation of Figure 1(d) is more efficient than the one in Figure 1(c) because we avoid the computation of the linear expression $|f(x,y)| - |f(x,x)|$ and as well as some intermediate term comparisons. ❑

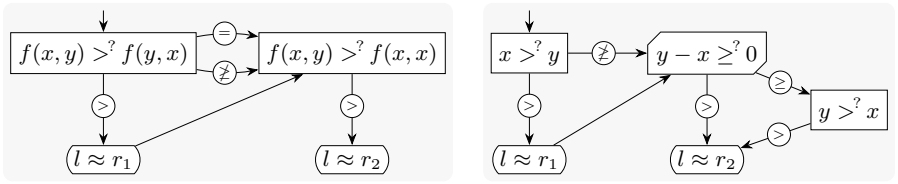
Example 1 shows that interleaving term comparisons and positivity checks can simplify, and hence speed up, term ordering checks. Let us now give formal definitions. We will use standard graph-theoretic notions related to directed acyclic graphs (dags), trees, and paths in a graph.

Definition 1 (Term Ordering Diagram – TOD). A *term ordering diagram* (TOD) is a directed acyclic graph $\mathcal{T}$ which contains five kinds of nodes:

(1) A single *root node*, so that every node is reachable from the root node. The root node has one outgoing edge.
(2) A single *exit node*, reachable from every node.
(3) A *term comparison node* is labeled by a term comparison $s >^? t$ and has three outgoing edges, labeled by $>$, $=$, and $\not\geq$.
(4) A *positivity check node* is labeled by a positivity check $e \geq^? 0$ and has three outgoing edges, labeled by $>$, $\geq$, and $\not\geq$.
(5) A *success node* is labeled by an equality $l \approx r$ and has one outgoing edge.❑

We collectively refer to the term comparison nodes and positivity check nodes as *evaluation nodes*. The idea is that we evaluate the substitution in these nodes and follow the node label resulting from the evaluation. Note that the exit node and its incoming edges are usually omitted when displaying a TOD (as in Figure 1). To retrieve equalities towards solving the post-ordering problem, we traverse a TOD as follows.

Definition 2 (TOD Traversal). Let σ be a query substitution and $\mathcal{T}$ a TOD. The $\mathcal{T}$ *traversal for* σ is the path in $\mathcal{T}$ from its root to its exit node so that:



(a) (b)

Fig. 2. Retrieving multiple equalities, where $l \stackrel{\text{def}}{=} f(x,y)$, $r_1 \stackrel{\text{def}}{=} f(y,x)$ and $r_2 \stackrel{\text{def}}{=} f(x,x)$.

(1) If the path contains a term comparison node labeled by $s >^? t$, then the path also contains the outgoing edge of this node labeled by $\sigma(s >^? t)$.
(2) If the path contains a positivity check node labeled by $e \geq^? 0$, then the path also contains the outgoing edge of this node labeled by $\sigma(e \geq^? 0)$.

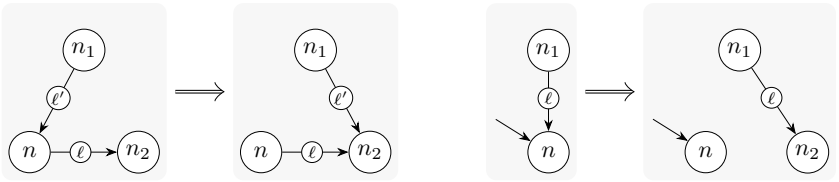
The *success set of the $\mathcal{T}$ traversal for σ* is the set of labels of all success nodes on this path; we also refer to it as *the success set of σ in $\mathcal{T}$*. For any non-exit node in a traversal, we refer to its *next edge* and *next node*.

For a node n labeled by c, where c is either a term comparison or a positivity check, we say that the node n *forces an edge label ℓ*, if for every query substitution σ, if the $\mathcal{T}$ traversal for σ reaches n, then $\sigma(c) = \ell$. ❑

We use a TOD to perform term ordering checks only when it is necessary. Key to this are TOD transformations (Section 4): given a TOD $\mathcal{T}_1$, we transform $\mathcal{T}_1$ into an equivalent TOD $\mathcal{T}_2$, so that $\mathcal{T}_2$ can be traversed faster, making ordering checks cheaper. The efficiency gain in $\mathcal{T}_2$ traversal comes by replacing $\mathcal{T}_1$ checks with less expensive ones or removing redundant checks of $\mathcal{T}_1$. Importantly, our TOD transformations on $\mathcal{T}_1$ are performed while we traverse $\mathcal{T}_1$ for a specific query substitution, as shown in Example 2.

Example 2. Consider again $l \stackrel{\text{def}}{=} f(x,y)$, $r_1 \stackrel{\text{def}}{=} f(y,x)$ and $r_2 \stackrel{\text{def}}{=} f(x,x)$. Let σ be a query substitution. Figure 2(a) shows the sequence of steps for retrieving $l \approx r_1$ if $(l \succ r_1)\sigma$, and then retrieving $l \approx r_2$ if $(l \succ r_2)\sigma$. Similarly to Figures 1(a) and (c), we can modify Figure 2(a) into its more efficient version displayed in Figure 2(b). However, in Figure 2(b) we implement stronger TOD modifications: in the (second) retrieval of $l \approx r_2$ we use information from the (first) retrieval of $l \approx r_1$. For example, if $x\sigma = y\sigma$, the evaluation of $x >^? y$ in Figure 2(b) results in neither $l \approx r_1$ nor $l \approx r_2$ retrieved. This evaluation is more efficient compared to its equivalent version in Figure 2(a) where both $f(x,y) >^? f(y,x)$ and $f(x,y) >^? f(x,x)$ have to be evaluated to obtain the same result. ❑

Examples 1–2 show that *we transform a TOD while we are traversing it.* This is done for two reasons. (i) First, term comparisons may involve expensive computations, for example, when we deal with linear expressions (as shown in Example 1). When we compute a linear expression, we modify the TOD to have a corresponding positivity check node, so that on the next traversal, if we reach the same node, we do not compute the linear expression again. (ii) Second, our



(a) (b)

Fig. 3. Generic transformations on TODs: (a) *redundant node removal*, where node n forces label ℓ; and (b) *node replication*, where n_2 is an exact copy of a non-exit node n that has multiple incoming edges.

transformations are *lazy* – we only modify nodes in a TOD when we reach them during traversal. The reason for this is that modifying TODs is expensive due to KBO/LPO checks; hence, these modifications should not be performed for nodes that will never be visited. Despite modifying TODs, we do not change their semantics: our TOD transformations (Section 4) yield equivalent TODs.

Definition 3 (TOD Equivalence). Two TODs $\mathcal{T}_1$ and $\mathcal{T}_2$ are *equivalent* if for every substitution σ, the success sets of σ in $\mathcal{T}_1$ and $\mathcal{T}_2$ coincide. ❑

4 TOD Transformations

We explain our TOD transformations via a sequence of subgraph replacements and show how TOD equivalences are preserved. Replacements are performed while visiting specific nodes n in the TOD, depending on the node label. As such, some replacements are *specific to KBO/LPO* constraints (Figures 4–5), whereas others are *generic* (Figure 3), independent of the ordering. Figure 3 summarizes our generic TOD transformations, as detailed below:

- Figure 3(a) corresponds to a *redundant node removal*, where n forces ℓ. Simply removing nodes might violate TOD properties: as n may have no incoming edges, the existence of a single root node in a TOD is not fulfilled. In such cases, we also remove n from the TOD. Generally, if a redundant node removal introduces a node with no incoming edges, we repeatedly remove such nodes.
- Figure 3(b) shows a *node replication*, where n is a non-exit node with multiple incoming edges and n_2 is the exact copy of n with the same outgoing edges.

Our *KBO and LPO transformations* on TODs are shown Figures 4 and 5, respectively. Here, we denote by s a term $f(s_1, \ldots, s_k)$, also written as $f(\bar{s})$, with $k \geq 0$. Similarly, t is a term $g(t_1, \ldots, t_m)$, also written as $g(\bar{t})$, with $m \geq 0$. Note that in Figure 5, if $f \gg g$ and $m = 0$, the node n_1 replaces the top node; conversely, in case of $g \gg f$ and $k = 0$, the node n_3 replaces the top node; finally, if $f = g$ and $k = 0$, then the node n_2 replaces the top node.

Theorem 1 (Correctness of TOD Transformations). *(1) Every sequence of TOD transformations terminates. (2) Every TOD transformation of Figures 3–5 preserves TOD equivalence.*

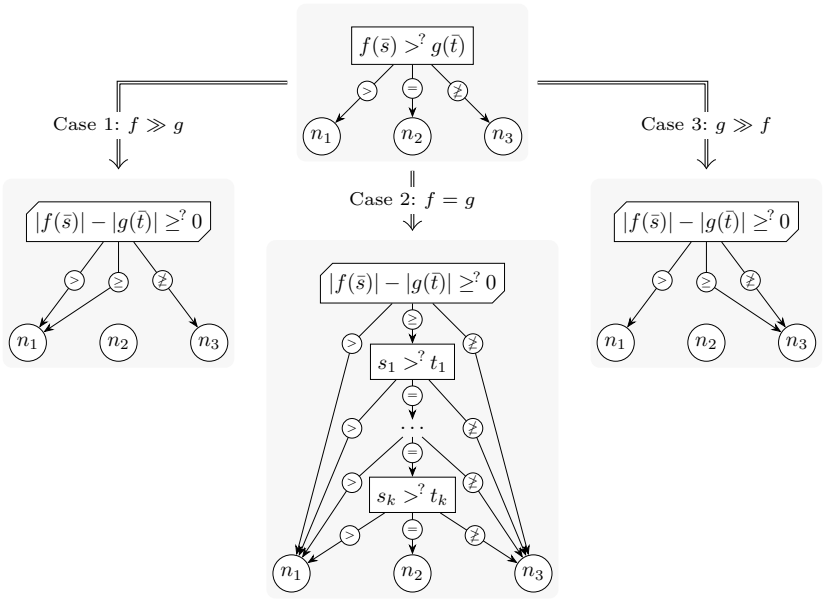


Fig. 4. KBO transformations on TODs.

Proof. (1) For proving *termination*, we introduce a well-founded order on TODs and show that every transformation replaces a TOD by a smaller one. In the proof, we use (well-founded) finite multiset extensions of (well-founded) orders.

We first introduce a mapping μ from nodes to finite multisets of terms as:

(i) If n is a term comparison node $s >^? t$, then $\mu(n)$ is the multiset $\{s, t\}$.
(ii) For any other node n, we define $\mu(n)$ to be the empty multiset.

Let us also define an order $>_\mu$ on nodes as follows: $n_2 >_\mu n_1$ if $\mu(n_2)$ is greater than $\mu(n_1)$ in the multiset extension of the order $\succ$ on terms. Note that $>_\mu$ is well-founded, since μ embeds it in the multiset extension of a well-founded order.

For every path π in a TOD, we denote by $\mu(\pi)$ the multiset consisting of elements $\mu(n)$ for all nodes n in π. We define an ordering, also denoted by $>_\mu$, on paths as follows: $\pi_2 >_\mu \pi_1$ if $\mu(\pi_2)$ is greater than $\mu(\pi_1)$ in the multiset extension of the order $>_\mu$ on nodes. Again, $>_\mu$ is well-founded, since it can be embedded to the multiset extension of a well-founded order.

Finally, for a TOD $\mathcal{T}$, we denote by $\mu(\mathcal{T})$ the multiset consisting of all multisets $\mu(\pi)$, where π is a path from the root node in $\mathcal{T}$. We also define an ordering $>_\mu$ on TODs by letting $\mathcal{T}_2 >_\mu \mathcal{T}_1$ if $\mu(\mathcal{T}_2) > \mu(\mathcal{T}_1)$. Using the same arguments as before, we conclude that $>_\mu$ on TODs is well-founded, too.

For every transformation apart from node replication that changes a TOD $\mathcal{T}$ to a TOD $\mathcal{T}'$, we have that $\mathcal{T} >_\mu \mathcal{T}'$, which implies termination. The proof is by routine inspection of transformations. For example for the case $f = g$ of the LPO transformations (last transformation of Figure 5), we replace a path with a term

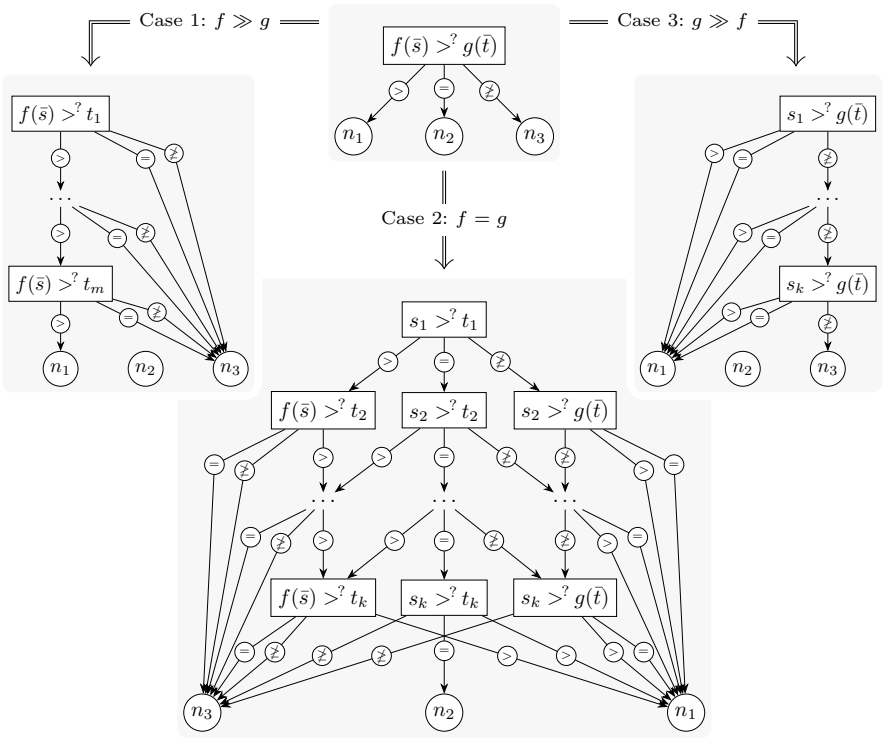


Fig. 5. LPO transformations on TODs.

comparison node $f(\bar{s}) >^? f(\bar{t})$ by a finite number of paths, so that every new term comparison node on these paths contains a comparison of a pair of terms strictly smaller in the multiset order than the multiset $\{f(\bar{s}), f(\bar{t})\}$. Another example is the redundant node removal of Figure 3. This transformation replaces on some paths nodes n_1, n, n_2 by n_1, n_2, which results in a smaller multiset.

Finally, we note that node replication results in a TOD containing exactly the same multiset of paths, but it can only be applied a finite number of times. *(2) TOD equivalence.* The TOD transformations of Figure 4–5 preserve TOD equivalence by properties of KBO/LPO. Redundant node removal in Figure 3(a) ensures equivalence by the "forces" relation (Definition 2). Node replication in Figures 3(b) preserves equivalence since the set of TOD paths does not change. ❑

5 TOD Retrieval and Maintenance

TOD is a data structure intended to be an index [24] for retrieval of equalities, which become ordered by the query substitution. In this section, we describe the three main operations on the TOD index: *insertion, deletion, and retrieval.* An

interesting feature of TODs is that the main modifications of them occur not during insertion but during retrieval. As mentioned in Section 3, we modify the TOD $\mathcal{T}$ lazily, when we traverse it for a query substitution σ, resulting in an equivalent TOD $\mathcal{T}'$. The new TOD $\mathcal{T}'$ is less expensive to traverse, since it either removes checks performed at nodes (redundant node removal) or replaces them with simpler ones (all transformations of Figures 4–5). Subsequent retrievals are then performed on the TOD $\mathcal{T}'$ instead of $\mathcal{T}$.

Index operations. The *insertion* of an equality $l \approx r$ is simple: we insert a term comparison node just before the exit node, as shown in Figure 6.

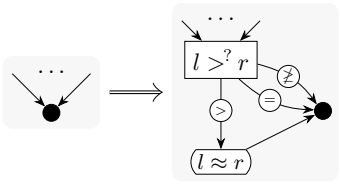


Fig. 6. Insertion of $l \approx r$

The *deletion* of an equality $l \approx r$ is not performed. Instead of doing the deletion, we simply memorize that $l \approx r$ is to be deleted. In practice, we do not even have to do this, since the clause containing $l \approx r$ will be marked as deleted anyhow.

A *retrieval* from a TOD consists of traversal, possibly interleaved with TOD transformations (Section 4). We next describe our TOD retrieval algorithm.

TOD retrieval algorithm. We introduce a notion meant to capture sufficient conditions for the (expensive) "forced" relation [5] , in extension of Definition 2.

Definition 4 (Forcing function). Let σ be a substitution. We call a path π a *σ-path*, if for every edge from a node n to a node n' in it labeled by ℓ, (1) if n is a term comparison node $s >^? t$, then $\sigma(s >^? t) = \ell$, and (2) if n is a positivity check node $e \geq^? 0$, then $\sigma(e \geq^? 0) = \ell$. We call a partial function F from paths to edge labels a *forcing function* if it has the following property. For every substitution σ and every σ-path $\pi = n_0, \ldots, n_i, n_{i+1}$ from the root, if $F(n_0, \ldots, n_i) = \ell$, then the edge in π from n_i to n_{i+1} is labeled with ℓ. ❑

Our retrieval algorithm is parametrized by a forcing function F. During the retrieval, we mark some nodes *visited*. Once a node n is marked visited, the path from the root to n will never change. Initially, all nodes in a TOD are unvisited. Let $\mathcal{T}$ be a TOD and σ a substitution. The traversal starts in the successor of the root node and repeatedly applies steps shown in Figure 7. When we describe the steps, we assume that during the traversal we already followed a path $\pi = n_0, n_1, \ldots, n_k, n$ leading to the current node n.

Let us now establish some properties of the algorithm, proving its correctness.

Lemma 1. *If n is a visited node, there is a single path from the root to n.*

Proof. When we make a node visited at step 3c of Figure 7, by induction we can assume that n_k already has this property. Since after this step there is only one edge to n, and this edge is from n_k, n satisfies this property. None of the other steps adds an incoming edge to a visited node or changes the content of a visited node, so $n_0, \ldots, n_k, n$ remains the only path to n. ❑

[5] We discuss forcing functions implemented in Vampire in the extended version of this paper [5].

1. If n is the exit node, we terminate and return the success set of σ for $\mathcal{T}$ (the set of all non-deleted equalities in success nodes visited during the traversal).
2. If n is a success node, we set n to the successor node of n. Alternatively, if we are interested in only one candidate, we can terminate once we reach the first success node with a non-deleted equality.
3. Let n be an unvisited evaluation node. If n has more than one incoming edge, we first apply node replication so that the only incoming edge to n is from n_k. Then,
   (a) If $F(\pi) = \ell$, we apply the transformation of Figure 3 (a).
   (b) If any of the transformations of Figures 4 and 5 applies to n, we apply this transformation.
   (c) Otherwise, we mark n visited.
4. If n is a visited term comparison node containing $s >^? t$, we follow the edge $\sigma(s >^? t)$ from n.
5. If n is a visited positivity check node containing $e \geq^? 0$, we follow the edge $\sigma(e \geq^? 0)$ from n.

Fig. 7. Retrieval algorithm steps.

Lemma 2. *Step 3a of Figure 7 transforms the TOD into an equivalent one.*

Proof. By Lemma 1, there is only one path from the root to n. By Definition 4, it follows that n forces ℓ, so this step is a special case of redundant mode removal from Figure 3(a), which preserves TOD equivalence by Theorem 1. ❑

Lemma 3 (Termination). *The retrieval algorithm terminates.*

Proof. Straightforward by Lemma 2 and Theorem 1. Indeed, all transformations made during the retrieval are special cases of TOD transformations, so we can only make a finite number of them. All other steps of the algorithm either change the current node to its successor in the TOD or mark an unvisited node as visited. Since any TOD is a dag, we can only make a finite number of such steps. ❑

Lemma 4 (Correctness). *The TOD $\mathcal{T}'$ resulting from the retrieval is equivalent to the TOD $\mathcal{T}$ before the retrieval.*

Proof. By Lemma 2, step 3a of Figure 7 is a special case of redundant node removal. All other transformations in Figure 7 are special cases of TOD transformations, which preserve equivalence by Theorem 1. ❑

Example 3. Consider the TOD in Figure 1(a). We perform retrieval on this TOD using a KBO with constant weight function and a query substitution σ such that $x\sigma = y\sigma$. The retrieval steps are shown in Figure 8. Initially, all nodes in the TOD are unvisited. We highlight the path up to the current node in each subdiagram with blue, and denote visited nodes with a red striped background. Note that the last node on the blue path is the current node n from the algorithm.

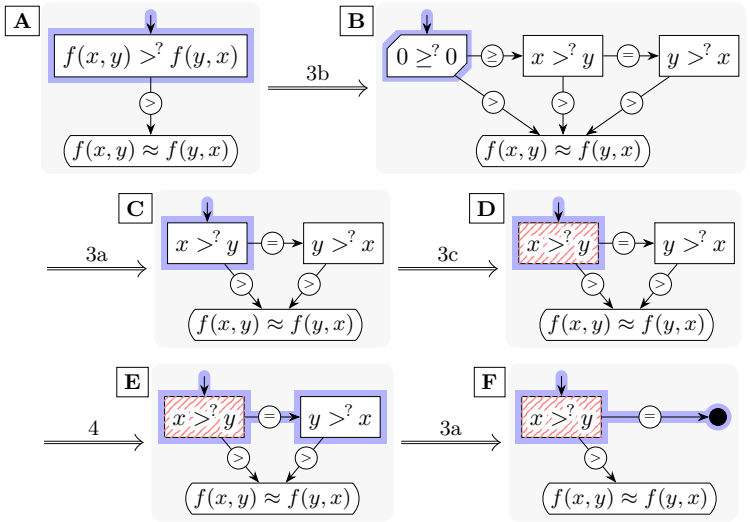


Fig. 8. Retrieval with a query substitution σ such that $x\sigma = y\sigma$.

1. Starting with sub-diagram **A** of Figure 8, we use step 3b, applying case 2 of the KBO transformations (see Figure 4).
2. In sub-diagram **B**, for the current path π we have that $F(\pi)$ equals $\geq$, so we apply redundant node removal (step 3a).
3. In sub-diagram **C**, the current node does not admit any transformations, so we mark it visited (step 3c).
4. In sub-diagram **D**, $\sigma(x >^? y)$ is $=$, so we follow the edge labeled $=$ (step 4).
5. In sub-diagram **E**, for the current path π we get that $F(\pi)$ equals $=$, due to $\sigma(x >^? y)$ being $=$. We apply redundant node removal and get to the exit node denoted explicitly with a black disc (step 3a).
6. In sub-diagram **F**, we return from the retrieval with no equalities (step 1).

Suppose that, before further traversals, we insert the equality $f(x,y) \approx f(x,x)$ into the TOD in sub-diagram **F** of Figure 8. The resulting TOD is shown in sub-diagram **G** of Figure 9, with a new term comparison node labeled $f(x,y) >^? f(x,x)$ and a new success node labeled $f(x,y) \approx f(x,x)$. The $>$ edge of the new term comparison node is connected to the new success node. The $=$ and $\not\geq$ edges of the new term comparison node, and the outgoing edge of the success node are connected to a new exit node (not shown). We traverse the TOD in sub-diagram **G** with a query substitution σ such that $x\sigma \succ y\sigma$.

1. Starting from sub-diagram **G** of Figure 9, we follow the edge $>$ (step 4), mark the success node labeled $f(x,y) \approx f(y,x)$ visited (step 3c) and follow its outgoing edge (step 2).
2. In sub-diagram **H**, we apply Case 2 of the KBO transformations (step 3b) and mark the current node labeled $y - x \geq^? 0$ visited (step 3c).

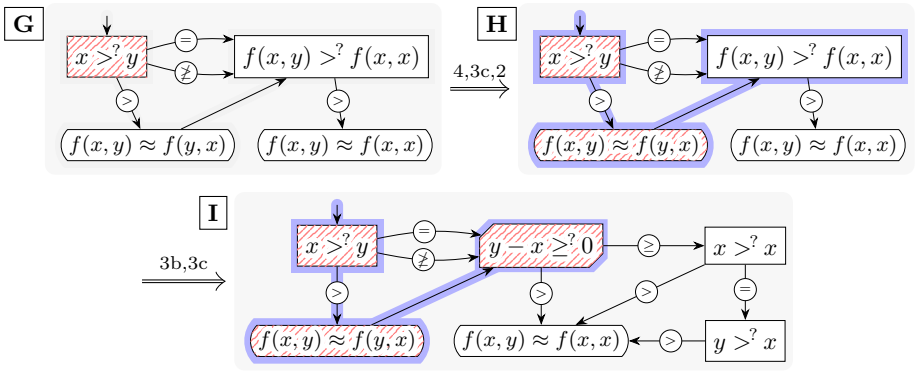


Fig. 9. Result of insertion of $f(x,y) \approx f(x,x)$ into the TOD of in sub-diagram **F** of Figure 8, and retrieval steps from top-left TOD with query substitution σ s.t. $x\sigma \succ y\sigma$.

3. In sub-diagram **I**, we follow the $\not\geq$ edge of the current node (step 5) and get to the exit node.
4. We exit with a single equality $f(x,y) \approx f(y,x)$ (step 1). ❑

6 Evaluation

We implemented our equality retrieval approach using TOD in Vampire.

Prover setup. We considered twelve configurations of Vampire options for reasoning using TOD, by taking Discount [2] or Otter [16] as saturation algorithm (`-sa discount/otter`), KBO or LPO as term order (`-to kbo/lpo`), and TOD variants using the new option `-fdtod` with values `off/on/shared` [6] for using TODs in forward demodulation, where:

(1) `off` does not use TODs,
(2) `on` uses a separate TOD for each forward demodulator (see e.g. Figure 1),
(3) `shared` uses a shared TOD for each set of forward demodulators with the same left-hand side (see e.g. Figure 2).

Benchmarks. We used version 8.2.0 of the TPTP library [26], which contains 25474 problems. We ignored problems where TODs are not used at all, resulting in 9310 problems used for evaluation.

Hardware. We used compute nodes with AMD Epyc 7502 2.5GHz processors and 1TB RAM. Each benchmark run relied on a single core and 16GB of memory.

[6] The option values `on` and `shared` are implemented in the `master` and `term-ordering-diagrams` branches of Vampire, respectively.

Table 1. Number of problems solved by Vampire within 60s. The numbers in parentheses show the number of problems lost and won by TOD variants compared to `off`.

		`off`	`on`	`shared`
Otter	KBO	3250 (-0, +0)	3258 (-0, +8)	3261 (-0, +11)
	LPO	3064 (-0, +0)	3083 (-1, +20)	3092 (-2, +30)
Discount	KBO	3182 (-0, +0)	3194 (-1, +13)	3196 (-1, +15)
	LPO	3016 (-0, +0)	3044 (-1, +29)	3058 (-1, +43)

Experimental summary. Table 1 shows the number of problems solved by Vampire with a 60 seconds timeout. Our approach using TODs is better, regardless of the term ordering (`to`) or the saturation algorithm (`sa`). Further, Vampire `shared` proves theorems significantly faster than `off`.[7]

Experiments using TODs seem to reach the memory limit in more instances than without TODs. For example in `-sa otter -to lpo`, 44 problems reach a memory limit with `shared`, whereas only 14 do so when TODs are `off`. However, Vampire `-fdtod off` hits some difficult ordering checks and is unable to generate huge clauses that consume a lot of memory.

Table 2 shows the proportion of machine instructions spent on forward demodulation and on entire runs, showcasing that we reduce the solving time of the post-ordering problem. We used an instruction limit of 200×10^9 (approximately 60 seconds). We highlight the subset of results from Table 2 for problems in the UEQ (unit equality) category of TPTP in Table 3. Post-ordering checks in this category generally take up a more significant proportion of instructions. In some cases, when using LPO without TODs, these checks even dominate the number of instructions.

7 Related Work and Conclusion

We introduce the term ordering diagram (TOD) index to improve the efficiency of term ordering checks. Our evaluation using Vampire shows significant improvement over naive term ordering checks.

Decidability and complexity of problems related to KBO and LPO are studied in [18,9,10,20], complemented by efficient implementations [15,22,13]. In particular, in [22] each KBO ordering check is runtime specialized individually in one step. We improve upon this by preprocessing ordering checks completely lazily. Additionally, TODs allow algorithm specialization for an arbitrary number of sequential ordering checks, including also for LPO. The open problem 9 of [19] is a generalization of the post-ordering problem. Confluence trees, developed to decide ground confluence of equational systems [1], are similar to TODs. Confluence trees have also been adapted to ground reducibility checking [12]. Shared rewriting avoids the post-ordering problem by caching the results of demodulations in shared terms [14]. Similarly to term indexing structures, the efficiency

[7] A more detailed evaluation can be found in the extended version of this paper [5].

Table 2. Number of instructions spent on post-ordering checks and forward demodulation compared to total instruction count. The first column describes the options, and the next three display the number of instructions performed on parts of Vampire. Columns 5–6 list the percentage of instructions spent on post-ordering checks and forward demodulation, and the last column gives the number of forward demodulations.

Options			# of instructions $\times 10^{12}$			Ratio to total		# of $\times 10^6$
			PostOrd.	Fw dem.	Total	PostOrd.	Fw dem.	Fw dem.
Otter	KBO	`off`	73	174	1258	5.8%	13.9%	2598
		`on`	34	142	1256	2.7%	11.3%	2714
		`shared`	18	116	1255	1.4%	9.2%	2813
	LPO	`off`	259	362	1292	20.0%	28.0%	1922
		`on`	154	279	1286	12.0%	21.7%	2064
		`shared`	98	213	1284	7.7%	16.6%	2238
Discount	KBO	`off`	91	314	1272	7.2%	24.7%	3044
		`on`	45	281	1258	3.6%	22.4%	3279
		`shared`	37	272	1269	3.0%	21.4%	3520
	LPO	`off`	214	417	1301	16.5%	32.1%	2376
		`on`	115	354	1283	8.9%	27.6%	2768
		`shared`	86	330	1294	6.7%	25.5%	3228

of term ordering diagrams is based on sharing information in indexed elements. However, the insertion retrieval operations for term ordering diagrams are done lazily, necessitated by the more expensive ordering comparison operations they support. To the best of our knowledge, our TOD approach gives the first algorithmic solution to efficient post-ordering checks.

Further work includes applying TODs to, for example, backward demodulation, constrained superposition, subsumption demodulation [4], ground reducibility [12], or ground joinability [3]. Extending our framework to other simplification orders, such as the weighted path order (WPO) [29], is another task for the future.

Acknowledgements. This research was funded in whole or in part by the ERC Consolidator Grant ARTIST 101002685, the ERC Proof of Concept Grant LEARN 101213411, the TU Wien Doctoral College SecInt, the FWF SpyCoDe Grant 10.55776/F85, the WWTF grant ForSmart 10.47379/ICT22007, and the Amazon Research Award 2023 QuAT.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Comon, H., Narendran, P., Nieuwenhuis, R., Rusinowitch, M.: Deciding the confluence of ordered term rewrite systems. ACM TOCL (2003). https://doi.org/10.1145/601775.601777

Table 3. Number of instructions spent on post-ordering checks and forward demodulation compared to total instruction count, within UEQ problems (1456 problems).

Options			# of instructions $\times 10^{12}$			Ratio to total		# of $\times 10^6$
			PostOrd.	Fw dem.	Total	PostOrd.	Fw dem.	Fw dem.
Otter	KBO	`off`	23.6	37.9	101.0	23.4%	37.6%	295.3
		`on`	10.8	28.1	100.3	10.8%	28.0%	342.4
		`shared`	5.1	18.8	99.9	5.1%	18.8%	382.9
	LPO	`off`	58.4	69.0	113.2	51.6%	60.9%	130.3
		`on`	33.9	50.6	110.0	30.8%	46.0%	158.2
		`shared`	20.0	34.1	111.2	18.0%	30.6%	215.5
Discount	KBO	`off`	26.8	70.3	112.6	23.8%	62.5%	726.5
		`on`	15.4	65.1	111.8	13.8%	58.3%	815.1
		`shared`	11.6	60.2	111.7	10.4%	54.0%	919.5
	LPO	`off`	63.0	95.2	121.5	51.8%	78.4%	420.3
		`on`	39.0	85.3	119.5	32.6%	71.4%	567.1
		`shared`	28.9	77.4	119.4	24.2%	64.8%	732.0

2. Denzinger, J., Kronenburg, M., Schulz, S.: DISCOUNT - A Distributed and Learning Equational Prover. Journal of Automated Reasoning (1997). https://doi.org/10.1023/A:1005879229581
3. Duarte, A., Korovin, K.: Ground Joinability and Connectedness in the Superposition Calculus. In: IJCAR (2022). https://doi.org/10.1007/978-3-031-10769-6_11
4. Gleiss, B., Kovács, L., Rath, J.: Subsumption Demodulation in First-Order Theorem Proving. In: IJCAR (2020). https://doi.org/10.1007/978-3-030-51074-9_17
5. Hajdu, M., Coutelier, R., Kovács, L., Voronkov, A.: Term Ordering Diagrams (extended) (2025). https://doi.org/10.48550/arXiv.2505.22181
6. Kamin, S., Lévy, J.J.: Two generalizations of the recursive path ordering. Unpublished manuscript (1980)
7. Knuth, D.E., Bendix, P.B.: Simple Word Problems in Universal Algebras. In: Automation of Reasoning 2. Springer (1983). https://doi.org/10.1007/978-3-642-81955-1_23
8. Korovin, K.: iProver — An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In: IJCAR (2008). https://doi.org/10.1007/978-3-540-71070-7_24
9. Korovin, K., Voronkov, A.: Knuth-Bendix Constraint Solving Is NP-Complete. In: ICALP (2001). https://doi.org/10.1007/3-540-48224-5_79
10. Korovin, K., Voronkov, A.: Orienting rewrite rules with the Knuth–Bendix order. Information and Computation (2003). https://doi.org/10.1016/S0890-5401(03)00021-X
11. Kovács, L., Voronkov, A.: First-Order Theorem Proving and Vampire. In: CAV. pp. 1–35 (2013). https://doi.org/10.1007/978-3-642-39799-8_1
12. Löchner, B.: A Redundancy Criterion Based on Ground Reducibility by Ordered Rewriting. In: CADE (2004). https://doi.org/10.1007/978-3-540-25984-8_2
13. Löchner, B.: Things To Know When Implementing LPO. International Journal on Artificial Intelligence Tools (2006). https://doi.org/10.1142/S0218213006002564
14. Löchner, B., Schulz, S.: An Evaluation of Shared Rewriting. In: IWIL (2001), `https://www.researchgate.net/publication/229079124`

15. Löchner, B.: Things to Know when Implementing KBO. Journal of Automated Reasoning (2006). https://doi.org/10.1007/s10817-006-9031-4
16. McCune, W., Wos, L.: Otter - the CADE-13 competition incarnations. Journal of Automated Reasoning (1997). https://doi.org/10.1023/A:1005843632307
17. Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Handbook of Automated Reasoning, chap. 7. Elsevier and MIT Press (2001). https://doi.org/10.1016/B978-044450813-3/50009-6
18. Nieuwenhuis, R.: Simple LPO constraint solving methods. Information Processing Letters (1993). https://doi.org/10.1016/0020-0190(93)90226-Y
19. Nieuwenhuis, R.: Invited Talk: Rewrite-Based Deduction and Symbolic Constraints. In: CADE (1999). https://doi.org/10.1007/3-540-48660-7_28
20. Nieuwenhuis, R., Rivero, J.M.: Practical Algorithms for Deciding Path Ordering Constraint Satisfaction. Information and Computation (2002). https://doi.org/10.1006/inco.2002.3146
21. Riazanov, A., Voronkov, A.: Partially Adaptive Code Trees. In: JELIA (2000). https://doi.org/10.1007/3-540-40006-0_15
22. Riazanov, A., Voronkov, A.: Efficient Checking of Term Ordering Constraints. In: CADE (2004). https://doi.org/10.1007/978-3-540-25984-8_3
23. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, Higher, Stronger: E 2.3. In: CADE (2019). https://doi.org/10.1007/978-3-030-29436-6_29
24. Sekar, R., Ramakrishnan, I., Voronkov, A.: Term Indexing. In: Handbook of Automated Reasoning, chap. 26. MIT Press (2001). https://doi.org/10.1016/B978-044450813-3/50028-X
25. Sutcliffe, G.: The CADE ATP System Competition - CASC. AI Magazine (2016). https://doi.org/10.1609/aimag.v37i2.2620
26. Sutcliffe, G.: Stepping Stones in the TPTP World. In: IJCAR (2024). https://doi.org/10.1007/978-3-031-63498-7_3
27. Voronkov, A.: The anatomy of Vampire: Implementing bottom-up procedures with code trees **15**(2), 237–265 (1995)
28. Vukmirovic, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making Higher-Order Superposition Work. Journal of Automated Reasoning (2022). https://doi.org/10.1007/s10817-021-09613-z
29. Yamada, A., Kusakari, K., Sakabe, T.: A unified ordering for termination proving. Science of Computer Programming (2015). https://doi.org/10.1016/j.scico.2014.07.009