SAT Solving for Argument Filterings*

Michael Codish[1], Peter Schneider–Kamp[2], Vitaly Lagoon[3], René Thiemann[2], and Jürgen Giesl[2]

[1] Dept. of Computer Science, Ben-Gurion University, Israel, mcodish@cs.bgu.ac.il
[2] LuFG Informatik 2, RWTH Aachen, Germany, {psk,thiemann,giesl}@informatik.rwth-aachen.de
[3] Dept. of Computer Science and Software Engineering, University of Melbourne, Australia, lagoon@cs.mu.oz.au

Abstract. This paper introduces a propositional encoding for lexicographic path orders in connection with dependency pairs. This facilitates the application of SAT solvers for termination analysis of term rewrite systems based on the dependency pair method. We address two main inter-related issues and encode them as satisfiability problems of propositional formulas that can be efficiently handled by SAT solving: (1) the combined search for a lexicographic path order together with an *argument filtering* to orient a set of inequalities; and (2) how the choice of the argument filtering influences the set of inequalities that have to be oriented. We have implemented our contributions in the termination prover AProVE. Extensive experiments show that by our encoding and the application of SAT solvers one obtains speedups in orders of magnitude as well as increased termination proving power.

1 Introduction

In recent work [5], Codish *et al.* introduce a propositional encoding of lexicographic path orders (LPO) and demonstrate that SAT solving can drastically speed up the solving of LPO termination problems. The key idea is that the encoding of a term rewrite system (TRS) $\mathcal{R}$ is satisfiable if and only if $\mathcal{R}$ is LPO-terminating and that each model of the encoding indicates a particular LPO which orients the rules in $\mathcal{R}$. However, lexicographic path orders on their own are too weak for many interesting termination problems and hence LPO is typically combined with more sophisticated termination proving techniques. One of the most popular and powerful such techniques is the *dependency pair* (DP) method [2]. Essentially, for any TRS the DP method generates a set of inequalities between terms. If one can find a well-founded order satisfying these inequalities, then termination is proved. A main advantage of the DP method is that it permits the use of orders which need not be monotonic. This allows the application of lexicographic path orders combined with *argument filterings*.

For every function symbol f, an argument filtering π specifies which parts of a term $f(\ldots)$ may be eliminated before comparing terms. As stated in [17],

* Supported by the Deutsche Forschungsgemeinschaft DFG under grant GI 274/5-1. This paper appeared in the *Proc. 13th LPAR*, LNAI, 2006.

"the dependency pairs method derives much of its power from the ability to use argument filterings to simplify constraints". However, argument filterings represent a severe bottleneck for the automation of dependency pairs, as the search space for argument filterings is enormous. In recent refinements of the DP method [11, 23], the choice of π also influences the set of *usable rules* which contribute to the inequalities that have to be oriented.

This paper extends the approach of [5] by providing a propositional encoding which combines the search for an LPO with the search for an argument filtering. This extension is non-trivial as the choice of an argument filtering π influences the structure of the terms in the rules as well as the set of rules which contribute to the inequalities that need to be oriented. The key idea is to combine all of the constraints on π which influence the definition of the LPO and the definition of the usable rules and to encode these constraints in SAT. This encoding captures the synergy between precedences on function symbols and argument filterings. In our approach there exist an argument filtering π and an LPO which orient a set of inequalities if and only if the encoding of the inequalities is satisfiable. Moreover, each model of the encoding corresponds to a suitable argument filtering and a suitable LPO which orient the inequalities.

After the necessary preliminaries on LPO and on the DP method in Sect. 2, Sect. 3 extends the approach of [5] to consider argument filterings. Sect. 4 shows how to extend this encoding to account for the influence of an argument filtering on the set of usable rules. In Sect. 5 we describe the implementation of our results in the termination prover AProVE [14] and provide extensive experimental evidence indicating speedups in orders of magnitude. We conclude in Sect. 6.

2 Preliminaries

This section briefly describes the starting points for the rest of the paper: propositional encodings for lexicographic path orders [5, 20] and the dependency pair framework [2, 12, 17]. We refer to [3] for further details on term rewriting.

We assume an algebra of terms constructed over given sets of symbols $\mathcal{F}$ and variables $\mathcal{V}$. Let $>_\mathcal{F}$ denote a (strict or non-strict) partial order on $\mathcal{F}$ (a so-called *precedence*) and let $\approx_\mathcal{F}$ denote the corresponding equivalence relation. We denote by $\sim$ the equality of terms up to equivalence of symbols. Observe that if $>_\mathcal{F}$ is strict then $\approx_\mathcal{F}$ and $\sim$ are the identity of symbols and terms respectively. Each precedence $>_\mathcal{F}$ on the symbols induces a lexicographic path order on terms.

Definition 1 (LPO [19]). *The lexicographic path order $\succ_{LPO}$ on terms induced by the partial order $>_\mathcal{F}$ is defined as $s = f(s_1, \ldots, s_n) \succ_{LPO} t$ if and only if one of the following holds:*

1. *$t = g(t_1, \ldots, t_m)$ and $s \succ_{LPO} t_j$ for all $1 \leq j \leq m$ and either*
   (i) $f >_\mathcal{F} g$ or (ii) $f \approx_\mathcal{F} g$ and $\langle s_1, \ldots, s_n \rangle \succ_{LPO}^{lex} \langle t_1, \ldots, t_m \rangle$; or
2. *$s_i \succsim_{LPO} t$ for some $1 \leq i \leq n$.*

Here $\succ_{LPO}^{lex}$ is the lexicographic extension of $\succ_{LPO}$ to tuples of terms and $\succsim_{LPO}$ is the union of $\succ_{LPO}$ and $\sim$.

The classical approach to prove termination of a TRS $\mathcal{R}$ is to find a *reduction order* $\succ$ which orients all rules $\ell \to r$ in $\mathcal{R}$ (i.e., $\ell \succ_{LPO} r$). A reduction order is an order which is well founded, monotonic, and stable (closed under contexts and substitutions). In practice, most reduction orders amenable to automation are *simplification orders* [7], i.e., they contain the embedding relation $\succ_{emb}$.

The LPO is one of the most prominent simplification orders and raises the associated decision problem: For terms s and t, is there a precedence $>_{\mathcal{F}}$ such that $s \succ_{LPO} t$ holds? This decision problem comes in two flavors: "strict-LPO" and "quasi-LPO" depending on whether $>_{\mathcal{F}}$ is required to be strict or not. Finding $>_{\mathcal{F}}$ such that $s \succ_{LPO} t$ is tantamount to solving a constraint obtained by unfolding the definition of $s \succ_{LPO} t$, cf. [6, 15].

As an example, let $\mathcal{F} = \{-, +, *\}$. Then there exists a strict precedence such that $-(x+y) \succ_{LPO} (-x) * (-y)$ if and only if the *partial order constraint* $(- >_{\mathcal{F}} *) \vee ((+ >_{\mathcal{F}} *) \wedge (+ >_{\mathcal{F}} -))$ has a solution. In [20] the authors show how such constraints can be encoded into propositional formulas. These formulas are satisfiable if and only if there exists a suitable partial order $>_{\mathcal{F}}$. A substantially improved encoding from such partial order constraints into propositional formulas is presented in [5].

It is well known that lexicographic path orders on their own are not very powerful for proving termination.

Example 2. Consider the following TRS $\mathcal{R}$ for division on natural numbers [2].

$$\begin{array}{rlrl}
\mathsf{minus}(x, \mathsf{0}) \to x & (1) & \mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) \to \mathsf{0} & (3)\\
\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{minus}(x, y) & (2) & \mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) & (4)
\end{array}$$

Rules (1) - (3) can easily be oriented using an LPO, but rule (4) cannot. To see this, observe that if we instantiate y by $\mathsf{s}(x)$, we obtain $\mathsf{quot}(\mathsf{s}(x), \mathsf{s}(\mathsf{s}(x))) \prec_{emb} \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, \mathsf{s}(x)), \mathsf{s}(\mathsf{s}(x))))$. Thus, no simplification order can show termination of $\mathcal{R}$. This drawback was the reason for developing more powerful approaches like the dependency pair method.

The dependency pair framework [12] is a modular reformulation and improvement of Arts and Giesl's dependency pair approach [2] which was also inspired by related work in [4, 17]. To ease readability, the following presentation is slightly simplified yet sufficient to state the contributions of this paper. For further details on the dependency pair framework see [12].

For a term rewrite system $\mathcal{R}$ over the symbols $\mathcal{F}$, the set of *defined* symbols $\mathcal{D}_{\mathcal{R}} \subseteq \mathcal{F}$ is the set of all root symbols of left-hand sides of $\mathcal{R}$. With each defined symbol $f \in \mathcal{D}_{\mathcal{R}}$ we extend the signature $\mathcal{F}$ by a fresh *tuple symbol* F. For each rule $f(s_1, \ldots, s_n) \to r$ in a term rewrite system $\mathcal{R}$ and for each subterm $g(t_1, \ldots, t_m)$ of r with $g \in \mathcal{D}_{\mathcal{R}}$, $F(s_1, \ldots, s_n) \to G(t_1, \ldots, t_m)$ is a dependency pair, intuitively indicating that a function call to f may lead to a function call to g. The set of dependency pairs of $\mathcal{R}$ is denoted $DP(\mathcal{R})$.

Example 3. Recall the term rewrite system from Ex. 2. The defined symbols are minus and quot and there are three dependency pairs:

$$\mathsf{MINUS}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{MINUS}(x, y) \quad (5)$$
$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{MINUS}(x, y) \quad (6)$$
$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{QUOT}(\mathsf{minus}(x, y), \mathsf{s}(y)) \quad (7)$$

The main result underlying the dependency pair method states that a term rewrite system $\mathcal{R}$ is terminating if and only if there is no infinite (minimal) $\mathcal{R}$-*chain* of its dependency pairs $DP(\mathcal{R})$ [2]. In other words, there is no infinite sequence of dependency pairs $s_1 \to t_1, s_2 \to t_2, \ldots$ from $DP(\mathcal{R})$ such that for all i there is a substitution σ_i where $t_i\sigma_i$ is terminating with respect to $\mathcal{R}$ and $t_i\sigma_i \to_{\mathcal{R}}^* s_{i+1}\sigma_{i+1}$. To prove absence of such infinite chains automatically, we consider so-called *dependency pair problems*. A dependency pair problem $(\mathcal{P}, \mathcal{R})$ is a pair of term rewrite systems $\mathcal{P}$ and $\mathcal{R}$ and poses the question: "Is there an infinite $\mathcal{R}$-chain of dependency pairs from $\mathcal{P}$?" The goal is to solve the dependency pair problem $(DP(\mathcal{R}), \mathcal{R})$ in order to determine termination of $\mathcal{R}$.

Termination techniques now operate on dependency pair problems and are called *DP processors*. Formally, a DP processor *Proc* takes a dependency pair problem as input and returns a new dependency pair problem which then has to be solved instead. A processor *Proc* is *sound* if for all dependency pair problems $(\mathcal{P}, \mathcal{R})$ where $\mathit{Proc}(\mathcal{P}, \mathcal{R}) = (\mathcal{P}', \mathcal{R})$, there is an infinite $\mathcal{R}$-chain of pairs from $\mathcal{P}'$ whenever there is an infinite $\mathcal{R}$-chain of pairs from $\mathcal{P}$. Soundness of a DP processor is required to prove termination and in particular, to conclude that there is no infinite $\mathcal{R}$-chain if $\mathit{Proc}(\mathcal{P}, \mathcal{R}) = (\varnothing, \mathcal{R})$.

So termination proofs in the DP framework start with the initial DP problem $(DP(\mathcal{R}), \mathcal{R})$. Then the DP problem is simplified repeatedly by sound DP processors. If one reaches the DP problem $(\varnothing, \mathcal{R})$, then termination is proved. In the following, we present one of the most important processors of the framework, the so-called *reduction pair processor*. Additional processors are described in [12].

For a DP problem $(\mathcal{P}, \mathcal{R})$, the reduction pair processor generates inequality constraints which should be satisfied by a *reduction pair* $(\succsim, \succ)$ [21] where $\succsim$ is reflexive, transitive, monotonic, and stable and $\succ$ is a stable well-founded order compatible with $\succsim$ (i.e., $\succsim \circ \succ \subseteq \succ$ or $\succ \circ \succsim \subseteq \succ$). However, $\succ$ need not be monotonic. A typical choice for a reduction pair $(\succsim, \succ)$ is to use simplification orders in combination with *argument filterings* [2] (we adopt notation of [21]).

Definition 4 (Argument Filtering). *An* argument filtering π *maps every n-ary function symbol to an argument position $i \in \{1, \ldots, n\}$ or to a (possibly empty) list $[i_1, \ldots, i_p]$ with $1 \leq i_1 < \cdots < i_p \leq n$. An argument filtering π induces a mapping from terms to terms:*

$$\pi(t) = \begin{cases} t & \text{if } t \text{ is a variable} \\ \pi(t_i) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } \pi(f) = i \\ f(\pi(t_{i_1}), \ldots, \pi(t_{i_p})) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } \pi(f) = [i_1, \ldots, i_p] \end{cases}$$

For a relation $\succ$ on terms, let $\succ^\pi$ be the relation where $s \succ^\pi t$ holds if and only if $\pi(s) \succ \pi(t)$. An argument filtering with $\pi(f) = i$ is called collapsing on f.

Arts and Giesl show in [2] that if $(\succsim, \succ)$ is a reduction pair and π is an argument filtering then $(\succsim^\pi, \succ^\pi)$ is also a reduction pair. In particular, we focus on reduction pairs of the form $(\succsim^\pi_{LPO}, \succ^\pi_{LPO})$ to prove termination of examples like Ex. 2 where the direct application of simplification orders fails.

The constraints generated by the reduction pair processor require that (a) all dependency pairs in $\mathcal{P}$ are weakly or strictly decreasing and, (b) all *usable* rules $\mathcal{U}(\mathcal{P}, \mathcal{R})$ are weakly decreasing. Here, a rule $f(\ldots) \to r$ from $\mathcal{R}$ is *usable* if f occurs in the right-hand side of a dependency pair from $\mathcal{P}$ or of a usable rule. In Ex. 2, the symbols occurring in the right-hand sides of the dependency pairs (5) - (7) are MINUS, QUOT, s, and minus. Therefore the minus-rules (1) and (2) are usable. Since the right-hand sides of the minus-rules do not contain additional symbols, these are in fact all of the usable rules. Hence, the quot-rules (3) and (4) are not usable.

As shown in [16, 23], under certain conditions on the reduction pair, Restriction (b) ensures that in chains $s_1 \to t_1, s_2 \to t_2, \ldots$ with $t_i\sigma_i \to^*_{\mathcal{R}} s_{i+1}\sigma_{i+1}$, we have $t_i\sigma_i \succsim s_{i+1}\sigma_{i+1}$. The required conditions hold in particular for any reduction pair constructed using simplification orders and argument filterings and specifically for $(\succsim^\pi_{LPO}, \succ^\pi_{LPO})$. Hence, the strictly decreasing pairs of $\mathcal{P}$ cannot occur infinitely often in chains. This enables the processor to delete such pairs from $\mathcal{P}$. In the following, for any term rewrite system $\mathcal{Q}$ and relation $\succ$, we denote $\mathcal{Q}_\succ = \{s \to t \in \mathcal{Q} \mid s \succ t\}$.

Theorem 5 (Reduction Pair Processor). *Let $(\succsim, \succ)$ be a reduction pair for a simplification order $\succ$ and let π be an argument filtering. Then the following DP processor Proc is sound.*

$$Proc(\mathcal{P}, \mathcal{R}) = \begin{cases} (\mathcal{P} \setminus \mathcal{P}_{\succ^\pi}, \mathcal{R}) & \textit{if } \mathcal{P}_{\succ^\pi} \cup \mathcal{P}_{\succsim^\pi} = \mathcal{P} \textit{ and } \mathcal{R}_{\succsim^\pi} \supseteq \mathcal{U}(\mathcal{P}, \mathcal{R}) \\ (\mathcal{P}, \mathcal{R}) & \textit{otherwise} \end{cases}$$

Example 6. For the term rewrite system of Ex. 2, according to Thm. 5 we search for a reduction pair solving the following inequality constraints.

$$\begin{array}{rlr} & \mathsf{MINUS}(\mathsf{s}(x), \mathsf{s}(y)) \ \underset{(\sim)}{\succ} \ \mathsf{MINUS}(x, y) & (8) \\ \mathsf{minus}(x, \mathsf{0}) \succsim x & \mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \ \underset{(\sim)}{\succ} \ \mathsf{MINUS}(x, y) & (9) \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \succsim \mathsf{minus}(x, y) & \mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \ \underset{(\sim)}{\succ} \ \mathsf{QUOT}(\mathsf{minus}(x, y), \mathsf{s}(y)) & (10) \end{array}$$

By Thm. 5, all dependency pairs corresponding to strictly decreasing inequalities (8) - (10) can be removed. To solve the inequalities we may take $(\succsim^\pi_{LPO}, \succ^\pi_{LPO})$ where $\pi(\mathsf{minus}){=}1$, $\pi(\mathsf{s}){=}\pi(\mathsf{MINUS}){=}\pi(\mathsf{QUOT}){=}[1]$, and where $\succsim_{LPO}$ and $\succ_{LPO}$ are induced by the partial order $\mathsf{QUOT} >_{\mathcal{F}} \mathsf{MINUS}$. For this choice, inequalities (8) - (10) are all strict and hence removed by the reduction pair processor. This results in the new DP problem $(\varnothing, \mathcal{R})$ which proves termination of Ex. 2.

We conclude this brief description of the dependency pair framework with a statement of the central *decision problem* associated with argument filterings, LPO, and dependency pairs:

> For a given dependency pair problem $(\mathcal{P}, \mathcal{R})$, does there exist a reduction pair $(\succsim^\pi_{LPO}, \succ^\pi_{LPO})$ for some argument filtering π and lexicographic path order induced by some partial order $>_{\mathcal{F}}$ such that all rules in $\mathcal{P}$ and in $\mathcal{R}$ are weakly decreasing and at least one rule in $\mathcal{P}$ is strictly decreasing?

In the following section we show how to encode constraints like "$s \succ^{\pi}_{LPO} t$" and "$s \succsim^{\pi}_{LPO} t$" as propositional formulas. Such an encoding enables us to encode the decision problem stated above as a SAT problem. Based on the solution of the SAT problem one can then identify the dependency pairs which can be removed from $\mathcal{P}$.

3 Encoding LPO and Argument Filtering

In this section we consider lexicographic path orders with argument filterings and the corresponding decision problem. Consider first a naive brute force approach. For any given argument filtering π we generate the formula

$$\bigwedge_{\ell \to r \in \mathcal{U}(\mathcal{P},\mathcal{R})} \pi(\ell) \succsim_{LPO} \pi(r) \quad \wedge \quad \bigwedge_{s \to t \in \mathcal{P}} \pi(s) \succsim_{LPO} \pi(t) \quad \wedge \quad \bigvee_{s \to t \in \mathcal{P}} \pi(s) \succ_{LPO} \pi(t) \quad (11)$$

The constraints "$\pi(s) \succsim_{LPO} \pi(t)$" and "$\pi(s) \succ_{LPO} \pi(t)$" can be encoded as described in Sect. 2. Then SAT solving can search for an LPO satisfying (11) for the given filtering π. However, this approach is hopelessly inefficient, potentially calling the SAT solver for each of the exponentially many argument filterings. Even if one considers the less naive enumeration algorithms implemented in [14] and [18], for many examples the SAT solver would be called exponentially often.

A contribution of this paper is to show instead how to encode the argument filterings into the propositional formula and delegate the search for an argument filtering to the SAT solver. In this way, the SAT solver is only called once with an encoding of Formula (11) and it can search for an argument filtering and for a precedence at the same time. This is clearly advantageous, since the filtering and the precedence highly influence each other.

So our goal is to encode constraints like "$s \succ^{\pi}_{LPO} t$" (or "$s \succsim^{\pi}_{LPO} t$") into propositional formulas such that every model of the encoding corresponds to a concrete filtering π and precedence $>_{\mathcal{F}}$ which satisfy "$s \succ^{\pi}_{LPO} t$" (or "$s \succsim^{\pi}_{LPO} t$"). We first provide an explicit definition which then provides the basis for specifying partial order and argument filtering constraints, satisfaction of which give "$s \succ^{\pi}_{LPO} t$" (or "$s \succsim^{\pi}_{LPO} t$"). The essential differences with Definition 1 are two: each of the two cases of Definition 1 is refined to consider the effect of π; and we define the weak version $\succsim^{\pi}_{LPO}$ of the order explicitly instead of just defining it via the equivalence on terms.

Definition 7 (LPO modulo π). *Let $>_{\mathcal{F}}$ be a (strict or non-strict) precedence and let π be an argument filtering on $\mathcal{F}$. Let x denote a variable.*

(I) The induced lexicographic path order $\succ^{\pi}_{LPO}$ on terms is defined as follows: $s = f(s_1, \ldots, s_n) \succ^{\pi}_{LPO} t$ if and only if one of the following holds:
  1. $t = g(t_1, \ldots, t_m)$ and
    (a) $\pi(g) = j$ and $s \succ^{\pi}_{LPO} t_j$; or
    (b) $\pi(f) = [i_1, ..., i_p]$, $\pi(g) = [j_1, ..., j_q]$, $s \succ^{\pi}_{LPO} t_j$ for all $j \in [j_1, \ldots, j_q]$, and either (i) $f >_{\mathcal{F}} g$ or
      (ii) $f \approx_{\mathcal{F}} g$ and $\langle s_{i_1}, \ldots, s_{i_p} \rangle \succ^{\pi,lex}_{LPO} \langle t_{j_1}, \ldots, t_{j_q} \rangle$; or

2. (a) $\pi(f) = i$ *and* $s_i \succ^{\pi}_{LPO} t$*; or*

   (b) $\pi(f) = [i_1, \ldots, i_p]$ *and for some* $i \in [i_1, \ldots, i_p]$, $(s_i \succsim^{\pi}_{LPO} t)$.

(II) For tuples of terms we define $\langle s_1, \ldots, s_n\rangle \succ^{\pi,lex}_{LPO} \langle t_1, \ldots, t_m\rangle$ *iff* $n > 0$ *and*

   (a) $m = 0$ *or*

   (b) $m > 0$ *and* $((s_1 \succ^{\pi}_{LPO} t_1) \vee ((s_1 \succsim^{\pi}_{LPO} t_1) \wedge \langle s_2, ..., s_n\rangle \succ^{\pi,lex}_{LPO} \langle t_2, ..., t_m\rangle))$.

(III) $\succsim^{\pi}_{LPO}$ *and* $\succsim^{\pi,lex}_{LPO}$ *are defined in an analogous way to* $\succ^{\pi}_{LPO}$ *and* $\succ^{\pi,lex}_{LPO}$*:*

   (a) replacing $\succ^{\pi}_{LPO}$ *by* $\succsim^{\pi}_{LPO}$ *in (I) 1(a) and 2(a); and*

   (b) adding the case $x \succsim^{\pi}_{LPO} g(t_1, \ldots, t_m)$ *iff* $\pi(g) = j$ *and* $x \succsim^{\pi}_{LPO} t_j$*, and the case* $x \succsim^{\pi}_{LPO} x$ *to (I); and*

   (c) replacing $\succ^{\pi,lex}_{LPO}$ *by* $\succsim^{\pi,lex}_{LPO}$ *in (I),(II) and adding* $\langle\,\rangle \succsim^{\pi,lex}_{LPO} \langle\,\rangle$ *to (II).*

It follows directly from Definitions 1, 4, and 7 that for all terms s and t we have $s \succ^{\pi}_{LPO} t \Leftrightarrow \pi(s) \succ_{LPO} \pi(t)$ and $s \succsim^{\pi}_{LPO} t \Leftrightarrow \pi(s) \succsim_{LPO} \pi(t)$.

The decision problem associated with Def. 7 is stated as follows: For terms s and t, does there exist a partial order $>_{\mathcal{F}}$ and an argument filtering π such that $s \succ^{\pi}_{LPO} t$ resp. $s \succsim^{\pi}_{LPO} t$ holds. This problem again comes in two flavors: "strict-LPO" and "quasi-LPO" depending on whether $>_{\mathcal{F}}$ is required to be strict or not. Our aim is to encode these decision problems as constraints on $>_{\mathcal{F}}$ and π, similar to the encoding of $s \succ_{LPO} t$ as a partial order constraint in Sect. 2. The difference is that now we have two types of constraints: constraints on the partial order $>_{\mathcal{F}}$ and constraints on the argument filtering π. To express constraints on argument filterings we use atoms of the following forms: "$\pi(f) = i$" to constrain π to map f to the value i; "$\pi(f) \ni i$" to constrain π to map f either to a list containing i or to i itself; and "$list(\pi(f))$" to constrain π to map f to a list. So "$list(\pi(f))$" means that π is not collapsing on f.

Each of the definitions (I) - (III) in Def. 7 induces an encoding to constraints on partial orders and argument filterings. In the following definition, we illustrate the encoding of $s \succ^{\pi}_{LPO} t$ for the case of strict-LPO with argument filterings. The encoding for $s \succsim^{\pi}_{LPO} t$ and the encodings for quasi-LPO are defined in a similar way. In the following definition, τ_{1a}, τ_{1b} and τ_2 are the parts of the encoding corresponding to cases 1(a), 1(b) and 2(a-b) in Def. 7 (I).

Definition 8 (Encoding strict-LPO with Argument Filterings). *The strict-LPO encoding of* $s \succ^{\pi}_{LPO} t$ *is a mapping* τ *from pairs of terms* s *and* t *to constraints defined by the rules depicted in Fig. 1 (where* x *denotes a variable).*

Example 9. Consider the first arguments of QUOT in dependency pair (7). Using Def. 8, after simplification of conjunctions, disjunctions, and implications with *true* and *false* we obtain:

$$\begin{aligned}\tau(\mathsf{s}(x) \succ^{\pi}_{LPO} \mathsf{minus}(x, y)) = {} & (\pi(\mathsf{minus}){=}1 \wedge list(\pi(\mathsf{s})) \wedge \pi(\mathsf{s}) \ni 1) \\ & \vee (list(\pi(\mathsf{s})) \wedge list(\pi(\mathsf{minus})) \wedge (\mathsf{s} >_{\mathcal{F}} \mathsf{minus}) \wedge \\ & (\pi(\mathsf{minus}) \ni 1 \rightarrow list(\pi(\mathsf{s})) \wedge \pi(\mathsf{s}) \ni 1) \wedge \neg(\pi(\mathsf{minus}) \ni 2))\end{aligned}$$

Thus, $\mathsf{s}(x) \succ^{\pi}_{LPO} \mathsf{minus}(x, y)$ holds if and only if minus is collapsed to its first argument and s is not filtered or if s and minus are not collapsed, s is greater than minus in the precedence, the second argument of minus is filtered, and whenever minus keeps the first argument then s keeps the first argument, too.

Encoding I:

$$\tau(s \succ^{\pi}_{LPO} t) = \tau_{1a}(s \succ^{\pi}_{LPO} t) \ \bigvee\ \tau_{1b}(s \succ^{\pi}_{LPO} t) \ \bigvee\ \tau_{2}(s \succ^{\pi}_{LPO} t)$$

$$\tau_{1a}(x \succ^{\pi}_{LPO} t) {=} \tau_{1b}(x \succ^{\pi}_{LPO} t) {=} \tau_{2}(x \succ^{\pi}_{LPO} t) {=} \tau_{1a}(s \succ^{\pi}_{LPO} x) {=} \tau_{1b}(s \succ^{\pi}_{LPO} x) {=} \mathit{false}$$

$$\tau_{1a}(s \succ^{\pi}_{LPO} g(t_1,\ldots,t_m)) = \bigvee_{1 \le j \le m} \Big((\pi(g) = j) \bigwedge \tau(s \succ^{\pi}_{LPO} t_j) \Big) \ \text{for non-variable } s$$

$$\begin{array}{l} \tau_{1b}(f(s_1,\ldots,s_n) \succ^{\pi}_{LPO} g(t_1,\ldots,t_m)) = \mathit{list}(\pi(f)) \bigwedge \mathit{list}(\pi(g)) \bigwedge (f >_{\mathcal{F}} g) \bigwedge \\ \quad \bigwedge_{1 \le j \le m} (\pi(g) \supseteq j) \rightarrow \tau(f(s_1,\ldots,s_n) \succ^{\pi}_{LPO} t_j) \ \text{for } f \neq g \end{array}$$

$$\begin{array}{l} \tau_{1b}(f(s_1,\ldots,s_n) \succ^{\pi}_{LPO} f(t_1,\ldots,t_n) = \mathit{list}(\pi(f)) \bigwedge \\ \quad \tau(\langle s_1,\ldots,s_n\rangle \succ^{\pi,lex}_{LPO,f} \langle t_1,\ldots,t_n\rangle) \bigwedge \\ \quad \bigwedge_{1 \le j \le n} (\pi(f) \supseteq j) \rightarrow \tau(f(s_1,\ldots,s_n) \succ^{\pi}_{LPO} t_j) \end{array}$$

$$\begin{array}{l} \tau_{2}(f(s_1,\ldots,s_n) \succ^{\pi}_{LPO} t) = \bigvee_{1 \le i \le n} \Big((\pi(f) = i) \bigwedge \tau(s_i \succ^{\pi}_{LPO} t) \Big) \qquad \vee \\ \quad \left(\mathit{list}(\pi(f)) \wedge \bigvee_{1 \le i \le n} (\pi(f) \supseteq i) \wedge \tau(s_i \succsim^{\pi}_{LPO} t) \right) \end{array}$$

Encoding II:

$$\begin{array}{l} \tau(\langle s_i,\ldots,s_n\rangle \succ^{\pi,lex}_{LPO,f} \langle t_i,\ldots,t_n\rangle) = \mathit{false} \text{ if } n = 0 \text{ else} \\ \quad ((\pi(f) \supseteq i) \bigwedge \tau(s_i \succ^{\pi}_{LPO} t_i)) \bigvee \\ \quad \Big(\ (\ (\pi(f) \supseteq i) \rightarrow \tau(s_i \succsim^{\pi}_{LPO} t_i) \ ) \ \bigwedge \ \tau(\langle s_{i+1},\ldots,s_n\rangle \succ^{\pi,lex}_{LPO,f} \langle t_{i+1},\ldots,t_n\rangle) \Big) \end{array}$$

Fig. 1. Encoding LPO with Argument Filterings

We proceed to describe how partial order and argument filtering constraints are transformed into propositional logic. The propositional encoding of partial order constraints is presented in more detail in [5].

Let $|\mathcal{F}| = m$. The basic idea is to interpret the symbols in $\mathcal{F}$ as indices in a partial order taking finite domain values from the set $\{1,\ldots,m\}$. Each symbol $f \in \mathcal{F}$ is thus modeled as $\langle f_k,\ldots,f_1\rangle$ with f_k the most significant bit and $k = \lceil \log_2 m \rceil$. The binary value of $\langle f_k,\ldots,f_1\rangle$ represents the position of f in the partial order. Of course, $f_k,\ldots,f_1$ may be equal to $g_k,\ldots,g_1$ for $f \neq g$, if a (possibly strict) partial order imposes no order between f and g, or if a non-strict partial order imposes $f \approx_{\mathcal{F}} g$. Constraints of the form $(f >_{\mathcal{F}} g)$ or $(f \approx_{\mathcal{F}} g)$ on $\mathcal{F}$ are interpreted as constraints on indices and it is straightforward to encode them in k-bit arithmetic: A constraint of the form $(f \approx_{\mathcal{F}} g)$ is encoded in k bits by

$$\|(f \approx_{\mathcal{F}} g)\|_k = \bigwedge_{1 \le i \le k} (f_i \leftrightarrow g_i).$$

A constraint of the form $(f >_{\mathcal{F}} g)$ is encoded in k bits by

$$\|(f >_{\mathcal{F}} g)\|_k = \begin{cases} (f_1 \wedge \neg g_1) & \text{if } k = 1 \\ (f_k \wedge \neg g_k) \vee ((f_k \leftrightarrow g_k) \wedge \|(f > g)\|_{k-1}) & \text{if } k > 1 \end{cases}$$

To encode argument filtering constraints, we associate with each symbol $f \in \mathcal{F}$ of arity n the propositional variables $list_f$ (which is true if and only if π is not collapsing on f) and $arg_f^1, \dots, arg_f^n$ (which indicate which arguments of f remain after filtering by π). We impose for each $f \in \mathcal{F}$ of arity n a constraint of the form $\neg list_f \rightarrow \bigoplus_{1 \le i \le n} arg_f^i$ where $\bigoplus_{1 \le i \le n} arg_f^i$ specifies that exactly one of the variables arg_f^i is *true* and the rest are *false*. The argument filtering constraints are then encoded as follows: $\|list(\pi(f))\| = list_f$; $\|\pi(f) \ni i\| = arg_f^i$; and $\|\pi(f) = i\| = \neg list_f \wedge arg_f^i$.

Example 10. Consider the encoding in Ex. 9 which contains partial order constraints and argument filtering constraints. Using the above encoding for these constraints, we obtain the following propositional formula. Since there are only $m = 2$ symbols s and minus, we choose $k = 1$ and encode the partial order constraint ($\mathsf{s} >_{\mathcal{F}} \mathsf{minus}$) as ($\mathsf{s}_1 \wedge \neg\mathsf{minus}_1$).

$$\begin{aligned}
\|\tau(\mathsf{s}(x) \succ^{\pi}_{LPO} \mathsf{minus}(x,y))\| = {} & (\neg list_{\mathsf{minus}} \wedge arg^1_{\mathsf{minus}} \wedge list_{\mathsf{s}} \wedge arg^1_{\mathsf{s}}) \\
& \vee (list_{\mathsf{s}} \wedge list_{\mathsf{minus}} \wedge (\mathsf{s}_1 \wedge \neg \mathsf{minus}_1) \wedge \\
& (arg^1_{\mathsf{minus}} \rightarrow list_{\mathsf{s}} \wedge arg^1_{\mathsf{s}}) \wedge \neg arg^2_{\mathsf{minus}})
\end{aligned}$$

4 Argument Filterings and Usable Rules

Recent improvements of the DP method [11, 23] significantly reduce the number of rules required to be weakly decreasing in the reduction pair processor of Thm. 5. We first recapitulate the improved reduction pair processor and then adapt our propositional encoding accordingly.

The idea is that one can restrict the set of usable rules by taking the argument filtering into account: in right-hand sides of dependency pairs or rules, an occurrence of f in the i-th argument of g will never be the cause to introduce a usable f-rule if the argument filtering eliminates g's i-th argument. For instance, when taking $\pi(\mathsf{QUOT}) = [2]$ in Ex. 2, the right-hand side of the *filtered* dependency pairs do not contain minus anymore. Thus, no rule is considered usable. In Def. 11, we define these restricted usable rules for a term t (initially corresponding to the right-hand side of a dependency pair). Here, we make the TRS $\mathcal{R}$ explicit to facilitate a straightforward encoding in Def. 14 afterwards.

Definition 11 (Usable Rules modulo π [11, 23]). *Let $\mathcal{R}$ be a TRS and π an argument filtering. For any function symbol f, let $Rls_{\mathcal{R}}(f) = \{\ell \rightarrow r \in \mathcal{R} \mid$ root$(\ell) = f\}$. For any term t, the* usable rules *$\mathcal{U}_\pi(t, \mathcal{R})$ modulo π are given by:*

$$\begin{aligned}
\mathcal{U}_\pi(x, \mathcal{R}) &= \varnothing \quad \textit{for all variables } x \\
\mathcal{U}_\pi(f(t_1, \dots, t_n), \mathcal{R}) &= Rls_{\mathcal{R}}(f) \quad \cup \\
& \quad \textstyle\bigcup_{\ell \rightarrow r \in Rls_{\mathcal{R}}(f)} \ \mathcal{U}_\pi(r, \mathcal{R} \setminus Rls_{\mathcal{R}}(f)) \quad \cup \\
& \quad \textstyle\bigcup_{\pi(f) \ni i} \ \mathcal{U}_\pi(t_i, \mathcal{R} \setminus Rls_{\mathcal{R}}(f))
\end{aligned}$$

For a set of dependency pairs $\mathcal{P}$, let $\mathcal{U}_\pi(\mathcal{P}, \mathcal{R}) = \bigcup_{s \rightarrow t \in \mathcal{P}} \mathcal{U}_\pi(t, \mathcal{R})$.

We now refine the reduction pair processor of Thm. 5 to consider usable rules modulo π.

Theorem 12 (Reduction Pair Processor modulo π [23]). *Let $(\succsim, \succ)$ be a reduction pair for a simplification order $\succ$ and let π be an argument filtering. Then the following DP processor Proc is sound.*

$$Proc(\mathcal{P}, \mathcal{R}) = \begin{cases} (\mathcal{P} \setminus \mathcal{P}_{\succ^\pi}, \mathcal{R}) & \text{if } \mathcal{P}_{\succ^\pi} \cup \mathcal{P}_{\succsim^\pi} = \mathcal{P} \text{ and } \mathcal{R}_{\succsim^\pi} \supseteq \mathcal{U}_\pi(\mathcal{P}, \mathcal{R}) \\ (\mathcal{P}, \mathcal{R}) & \text{otherwise} \end{cases}$$

Example 13. Consider the following TRS (together with the minus-rules (1), (2))

$$\begin{aligned} \mathsf{ge}(x, \mathsf{0}) &\to \mathsf{true} && (12) & \mathsf{div}(x, y) &\to \mathsf{if}(\mathsf{ge}(x, y), x, y) && (15) \\ \mathsf{ge}(\mathsf{0}, \mathsf{s}(y)) &\to \mathsf{false} && (13) & \mathsf{if}(\mathsf{true}, \mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{s}(\mathsf{div}(\mathsf{minus}(x, y), \mathsf{s}(y))) && (16) \\ \mathsf{ge}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{ge}(x, y) && (14) & \mathsf{if}(\mathsf{false}, x, \mathsf{s}(y)) &\to \mathsf{0} && (17) \end{aligned}$$

The usable rules are the minus- and ge-rules since minus occurs in the right-hand side of the dependency pair $\mathsf{IF}(\mathsf{true}, \mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{DIV}(\mathsf{minus}(x, y), \mathsf{s}(y))$ resulting from rule (16) and ge occurs in the dependency pair $\mathsf{DIV}(x, y) \to \mathsf{IF}(\mathsf{ge}(x, y), x, y)$ resulting from rule (15). However, if one chooses an argument filtering with $\pi(\mathsf{DIV}) = [1]$ and $\pi(\mathsf{IF}) = [2]$, then the ge-rules are no longer usable since ge does not occur in the right-hand side of the filtered dependency pair $\mathsf{DIV}(x) \to \mathsf{IF}(x)$. Now Thm. 12 only requires the filtered minus-rules and the dependency pairs to be decreasing.

As demonstrated in [11, 23] and confirmed by the experiments described in Sect. 5, introducing argument filterings to the specification of usable rules results in a significant gain of termination proving power. However, Thm. 12 is not straightforward to automate using SAT solvers. The technique of Sect. 3 assumes a given set of inequalities which is then encoded to a propositional formula. The problem with Thm. 12 is that that the set of inequalities to be oriented depends on the chosen argument filtering. Hence, the search for an argument filtering should be combined with the computation of the usable rules. As discussed before, an enumeration of argument filterings is hopelessly inefficient. Therefore, we modify the encoding of the inequalities in Formula (11) to consider for every rule $\ell \to r \in \mathcal{R}$, the condition under which $\ell \to r$ is usable. Only under this condition one has to require the inequality $\pi(\ell) \succsim_{LPO} \pi(r)$. To this end, instead of encoding formula (11) we encode the following formula.

$$\underbrace{\bigwedge_{\ell \to r \in \mathcal{U}_\pi(\mathcal{P}, \mathcal{R})} \ell \succsim^\pi_{LPO} r}_{(a)} \;\wedge\; \underbrace{\bigwedge_{s \to t \in \mathcal{P}} s \succsim^\pi_{LPO} t}_{(b)} \;\wedge\; \underbrace{\bigvee_{s \to t \in \mathcal{P}} s \succ^\pi_{LPO} t}_{(c)} \qquad (11')$$

The subformulas (b) and (c) are identical to those in Formula (11) and are encoded as a conjunction and disjunction of encodings of the forms $\tau(s \succsim^\pi_{LPO} t)$ and $\tau(s \succ^\pi_{LPO} t)$ using Def. 8. The definition of the usable rules in Def. 11 now induces the following encoding of subformula (a) as a propositional formula $\omega(\mathcal{P},\mathcal{R})$.[4] As in Sect. 3, we use argument filtering constraints of the form "$\pi(f) \ni i$". Moreover, we introduce a new propositional variable u_f for every defined function symbol f of $\mathcal{U}(\mathcal{P},\mathcal{R})$ which indicates whether f's rules are usable.

Definition 14 (Encoding Usable Rules modulo Argument Filtering). *For a term t and a TRS $\mathcal{R}$ the formula $\omega(t,\mathcal{R})$ is defined as follows:*

$$\begin{array}{rll}
\omega(x,\mathcal{R}) = & \mathit{true} & \text{for } x \in \mathcal{V} \\
\omega(f(t_1,\ldots,t_n),\mathcal{R}) = & \bigwedge_{1\le i\le n} (\pi(f) \ni i \to \omega(t_i,\mathcal{R})) & \text{for } f \notin \mathcal{D}_{\mathcal{R}} \\
\omega(f(t_1,\ldots,t_n),\mathcal{R}) = & u_f \quad \wedge & \text{for } f \in \mathcal{D}_{\mathcal{R}} \\
& \bigwedge_{\ell\to r\in Rls_{\mathcal{R}}(f)} \omega(r, \mathcal{R} \setminus Rls_{\mathcal{R}}(f)) \quad \wedge & \\
& \bigwedge_{1\le i\le n} (\pi(f) \ni i \to \omega(t_i, \mathcal{R} \setminus Rls_{\mathcal{R}}(f))) &
\end{array}$$

For a set of dependency pairs $\mathcal{P}$, let

$$\omega(\mathcal{P},\mathcal{R}) = \left(\bigwedge_{s\to t\in\mathcal{P}} \omega(t,\mathcal{R})\right) \wedge \left(\bigwedge_{f\in\mathcal{D}_{\mathcal{U}(\mathcal{P},\mathcal{R})}} u_f \to \left(\bigwedge_{\ell\to r\in Rls_{\mathcal{R}}(f)} \tau(\ell \succsim^{\pi}_{LPO} r)\right)\right).$$

For a DP problem $(\mathcal{P},\mathcal{R})$ we encode the formula $(11')$. Every model of this encoding corresponds to a precedence $>_{\mathcal{F}}$ and an argument filtering π satisfying the constraints of the improved reduction pair processor from Thm. 12. Thus, we can now use SAT solving to automate Thm. 12 as well.

Example 15. Consider the TRS $\mathcal{R}$ from Ex. 13. Using the encoding of Def. 14, for $\mathcal{P} = DP(\mathcal{R})$ we obtain:

$$\begin{array}{l}
\omega(\mathcal{P},\mathcal{R}) = (\pi(\mathsf{DIV}) \ni 1 \to u_{\mathsf{minus}}) \wedge (\pi(\mathsf{IF}) \ni 1 \to u_{\mathsf{ge}}) \wedge \\
\quad (u_{\mathsf{minus}} \to (\tau(\mathsf{minus}(x,\mathsf{0}) \succsim^{\pi}_{LPO} x) \wedge \tau(\mathsf{minus}(\mathsf{s}(x),\mathsf{s}(y)) \succsim^{\pi}_{LPO} \mathsf{minus}(x,y)))) \wedge \\
\quad\quad (u_{\mathsf{ge}} \to (\tau(\mathsf{ge}(x,\mathsf{0}) \succsim^{\pi}_{LPO} \mathsf{true}) \wedge \tau(\mathsf{ge}(\mathsf{0},\mathsf{s}(y)) \succsim^{\pi}_{LPO} \mathsf{false}) \wedge \\
\quad\quad\quad \tau(\mathsf{ge}(\mathsf{s}(x),\mathsf{s}(y)) \succsim^{\pi}_{LPO} \mathsf{ge}(x,y))))
\end{array}$$

5 Implementation and Experiments

The propositional encodings for LPO with argument filterings and for the reduction pair processors described in Sect. 3 and 4 are implemented and integrated in the termination prover AProVE available from [9]. This Java implementation consists of the following main components: **(a)** An encoder from DP problems to formulas with partial order and argument filtering constraints (approx. 1700 lines). **(b)** A propositional encoder for partial order constraints following [5] and for argument filtering constraints (approx. 300 lines). **(c)** Interfaces to several SAT solvers (approx. 300 lines). In our experiments to evaluate the contributions of this paper, we applied the MiniSAT solver [8]. For the translation to conjunctive normal form (CNF) we used the implementation of Tseitin's algorithm [24] offered by SAT4J [22] - a freely available Java implementation of MiniSAT. Our implementation uses several optimizations to minimize encoding size:

[4] The definition of ω can easily be adapted to more advanced definitions of usable rules as well, cf. e.g. [2, 11, 13].

1. We apply basic simplification axioms for *true* and *false* as well as standard Boolean simplifications to flatten nested conjunctions and disjunctions.
2. When building the formulas top-down, at each point we maintain the sets of atomic constraints (partial order and argument filtering) that must be *true* and *false* from this point on. This information is then applied to simplify all constraints generated below (in the top-down process) and to prune the encoding process.
3. We memo and identify identical subformulas in the propositional encodings and represent formulas as directed acyclic graphs (or Boolean circuits) instead of trees. This decreases the size of the representation considerably. (The usefulness of sharing when solving LPO constraints was already discussed in [10].) For instance, consider the constraint from Ex. 9. Already in this tiny example, the subformula $list(\pi(\mathsf{s})) \wedge \pi(\mathsf{s}) \supseteq 1$ occurs twice, since it results from the encoding of both $\mathsf{s}(x) \succ_{LPO}^{\pi} x$ and $\mathsf{s}(x) \succ_{LPO}^{\pi} y$.

Optimization (2) typically reduces the number of propositional variables in the resulting CNF by a factor of at least 2. Optimizations (1) and (3) together further reduce the number of propositional variables by a typical factor of 10.

To evaluate our new SAT-based implementation, we performed extensive experiments to compare it with the corresponding methods in the current non-SAT-based implementations of AProVE [14] and of the Tyrolean Termination Tool (TTT) [18]. In the annual *International Competition of Termination Tools* 2004 and 2005 [1], AProVE and TTT were the two most powerful tools for termination analysis of term rewriting. For our experiments, both AProVE and TTT were configured to consider all argument filterings.[5]

We ran the three tools on all 773 TRSs from the *Termination Problem Data Base 2005.* This is the collection of examples from the annual competition of termination tools. It contains 99 TRSs that are known to be non-terminating and which serve as an error-checking mechanism. As expected, all three implementations fail to show termination of these TRSs. For the experiments, the TTT analyzer was applied via its web interface and ran on a Xeon 2.24GHz dual-CPU platform. The AProVE analyzer and our new SAT-based analyzer were run on an AMD Athlon 64 at 2.2 GHz.

Apart from the reduction pair processor, we also used the *dependency graph processor* [2, 12, 17], which is the other main processor of the dependency pair framework. This processor is used to split up dependency pair problems into smaller ones. As AProVE and TTT use slightly different techniques for estimating dependency graphs in the dependency graph processor and as they run on different machines, their performance is not directly comparable.

For a fair comparison of the three different implementations, we did not use any of the many other termination analysis techniques available in AProVE and TTT. In particular we did not use any techniques to preprocess the TRSs and we did not apply any other DP processors.

Tables 1 and 2 summarize the results using the DP processors based on

[5] TTT offers two algorithms to search for argument filterings. We used the "divide-and-conquer"-algorithm, since it is usually the more efficient one.

Thm. 5 and 12 respectively. The tools are indicated as: TTT, APR (AProVE) and SAT (AProVE with our SAT-based encoding). For each of the experiments we considered reduction pairs based on *strict*- and *quasi*-LPO. Each of the experiments was performed with a time-out of 60 seconds (corresponding to the way tools are evaluated in the annual competition) and with a time-out of 10 minutes. We indicate by "*Yes*", "*Fail*", and "*RL*" the number of TRSs for which proving termination with the given technique succeeds, fails, or encounters a resource limit (time-out or exhausts memory). Finally, we give the total time in seconds for analyzing all 773 examples. Individual runtimes and proof details are available from our empirical evaluation web site [9].

	LPO - 60sec t/o				LPO - 10min t/o				QLPO - 60sec t/o				QLPO - 10min t/o			
Tool	Yes	Fail	RL	Time	Yes	Fail	RL	Time	Yes	Fail	RL	Time	Yes	Fail	RL	Time
TTT	268	448	57	4202	269	465	39	28030	297	395	81	6241	297	408	68	43540
APR	310	358	105	6936	310	365	98	60402	320	331	122	7913	326	341	106	67764
SAT	327	446	0	82	327	446	0	82	359	414	0	183	359	414	0	183

Table 1. Strict–LPO (left) and Quasi-LPO (right) with the DP processor of Thm. 5

The comparison of the corresponding SAT-based and non-SAT-based configurations in Table 1 shows that the analyzers based on SAT solving with our proposed encoding are faster by orders of magnitude. Moreover, the power (i.e., the number of examples where termination can be proved) also increases substantially in the SAT-based configurations. It is also interesting to note that there are no time-outs in the SAT-based configurations, whereas the non-SAT-based configurations have many time-outs.[6]

	LPO - 60sec t/o				LPO - 10min t/o				QLPO - 60sec t/o				QLPO - 10min t/o			
Tool	Yes	Fail	RL	Time	Yes	Fail	RL	Time	Yes	Fail	RL	Time	Yes	Fail	RL	Time
APR	338	368	67	4777	341	383	49	33329	357	323	93	6100	359	336	78	49934
SAT	348	425	0	82	348	425	0	82	380	393	0	193	380	393	0	193

Table 2. Strict-LPO (left) and Quasi-LPO (right) with the DP processor of Thm. 12

Table 2 provides results using the improved reduction pair processor of Thm. 12. Again, the SAT-based configuration is much faster than the corresponding non-SAT-based one. The comparison with Table 1 shows that replacing the processor of Thm. 5 by the one of Thm. 12 increases power significantly and has no negative influence on runtimes.

In both tables, the comparison between strict- and quasi-LPO (of corresponding configurations) shows that quasi-LPO is more powerful but also slower than strict-LPO. However, for the SAT-based analyses, the overall runtimes are still extremely fast in comparison to the non-SAT-based configurations.

Table 3 highlights 5 examples which could not be solved by any tool in the termination competition 2005, whereas the SAT-based configuration proves termination for all 5 in a total of 4.3 seconds. In fact, except for the second example, neither TTT nor AProVE are able to prove termination within 10 minutes in their

[6] To evaluate the optimizations on p. 11, we also tested the SAT-based configuration with strict-LPO and the 10-minute time-out in a version where optimizations (2) and (3) are switched off. Here, the total runtime increases from 82 to 1968 seconds. Thus, optimizations (2) and (3) decrease total runtime by a factor of more than 20.

fully automatic mode (which uses many other termination techniques in addition to LPO and argument filtering). This demonstrates that our encoding advances the state of the art of automated termination analysis. The third example could be proven terminating by TTT or AProVE if they employed a strategy which applies LPO earlier. But due to efficiency considerations, both tools did not do this yet in their fully automatic mode. However, with the speed of our new SAT-based approach one can now develop strategies which try LPO and argument filtering as one of the first termination techniques. Since failure of LPO is now detected very quickly, one can still use other termination techniques afterwards.

The columns TTT, APR, and SAT indicate for the 3 tools the analysis times in seconds (including parsing, producing proofs, computing dependency graphs, etc.) and "t/o" indicates a 10 minute timeout. For each of the examples and tools, the time indicated is for the fastest configuration from those described in Tables 1 and 2. For the second and third example, TTT's "divide-and-conquer"-algorithm times out, but its "enumeration"-algorithm (which is usually less efficient) finds a solution within 10 minutes. Therefore, here the runtimes are given in brackets. The last four columns give details for the largest CNF which occurred during the termination proof with SAT (ranging over all dependency pair problems encountered). Columns 4 and 5 indicate the number of clauses and the number of literals of this CNF while Columns 6 and 7 indicate the time (in milliseconds) for encoding to propositional logic and for SAT solving.

Example	TTT	APR	SAT	# clauses	# literals	encod. time	SAT time
Ex26_Luc03b_Z	t/o	t/o	1.15	12462	32027	90	48
Ex2_Luc02a_C	(476.8)	t/o	0.69	8478	21200	137	20
Ex49_GM04_C	(25.8)	44.4	0.81	7040	17638	212	16
ExSec11_1_Luc02a_C	t/o	t/o	0.78	10968	28265	145	12
ExSec11_1_Luc02a_GM	t/o	t/o	0.87	19782	50608	155	72

Table 3. Five hard examples: SAT solving increases termination proving power

6 Conclusion

In [5] the authors demonstrate the power of propositional encoding and application of SAT solving to LPO termination analysis. This paper extends the SAT-based approach to consider the more realistic setting of dependency pair problems with LPO and argument filtering. The main challenge derives from the strong dependencies between the notions of LPO, argument filterings, and the set of rules which need to be oriented. The key to a solution is to introduce and encode in SAT all of the constraints originating from these notions into a single search process. We introduce such an encoding and through implementation and experimentation prove that it meets the challenge yielding speedups in orders of magnitude over existing termination tools as well as increasing termination proving power. To experiment with our SAT-based implementation and for further details on our experiments please visit our web page at `http://aprove.informatik.rwth-aachen.de/eval/SATLPO` [9].

References

1. Annual International Competition of Termination Tools. `http://www.lri.fr/~marche/termination-competition`.
2. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
3. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge, 1998.
4. C. Borralleras. *Ordering-based methods for proving termination automatically*. PhD thesis, Universitat Politècnica de Catalunya, Spain, 2003.
5. M. Codish, V. Lagoon, and P. J. Stuckey. Solving partial order constraints for LPO termination. In *Proc. RTA '06*, LNCS 4098, pages 4–18, 2006.
6. H. Comon, P. Narendran, R. Nieuwenhuis, and M. Rusinowitch. Decision problems in ordered rewriting. In *Proc. LICS '98*, pages 276–286, 1998.
7. N. Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1,2):69–116, 1987.
8. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT '03*, LNCS 2919, pages 502–518, 2004.
9. Empirical evaluation of "SAT Solving for Argument Filterings". `http://aprove.informatik.rwth-aachen.de/eval/SATLPO`.
10. T. Genet and I. Gnaedig. Termination proofs using GPO ordering constraints with shared term data structure. In *Proc. TAPSOFT '97*, LNCS 1214, pp. 249-260, 1997.
11. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Improving dependency pairs. In *Proc. LPAR '03*, LNAI 2850, pages 165–179, 2003.
12. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR '04*, LNAI 3452, pages 301–331, 2005.
13. J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proc. FroCoS '05*, LNAI 3717, pp. 216-231, 2005.
14. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the DP framework. *Proc. IJCAR '06*, LNAI 4130, pp. 281-286, 2006.
15. N. Hirokawa and A. Middeldorp. Tsukuba Termination Tool. In *Proc. RTA '03*, LNCS 2706, pages 311–320, 2003.
16. N. Hirokawa and A. Middeldorp. Dependency pairs revisited. In *Proc. RTA '04*, LNCS 3091, pages 249–268, 2004.
17. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2):172–199, 2005.
18. N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool. In *Proc. RTA '05*, LNCS 3467, pages 175–184, 2005.
19. S. Kamin and J. J. Lévy. Two generalizations of the recursive path ordering. Unpublished Manuscript, University of Illinois, IL, USA, 1980.
20. M. Kurihara and H. Kondo. Efficient BDD encodings for partial order constraints with application to expert systems in software verification. In *Proc. IEA/AIE '04*, LNCS 3029, pages 827–837, 2004.
21. K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In *Proc. PPDP '99*, LNCS 1702, pages 47–61, 1999.
22. SAT4J satisfiability library for Java. `http://www.sat4j.org`.
23. R. Thiemann, J. Giesl, and P. Schneider-Kamp. Improved modular termination proofs using dependency pairs. In *Proc. IJCAR '04*, LNAI 3097, pages 75–90, 2004.
24. G. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125. 1968. Reprinted in J. Siekmann and G. Wrightson (editors), *Automation of Reasoning*, vol. 2, pages 466-483, Springer, 1983.