Automated Termination Proofs with AProVE*

Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke

LuFG Informatik II, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany
{giesl|thiemann|psk}@informatik.rwth-aachen.de
spf@i2.informatik.rwth-aachen.de

Abstract. We describe the system AProVE, an automated prover to verify (innermost) termination of term rewrite systems (TRSs). For this system, we have developed and implemented efficient algorithms based on classical simplification orders, dependency pairs, and the size-change principle. In particular, it contains many new improvements of the dependency pair approach that make automated termination proving more powerful and efficient. In AProVE, termination proofs can be performed with a user-friendly graphical interface and the system is currently among the most powerful termination provers available.

1 Introduction

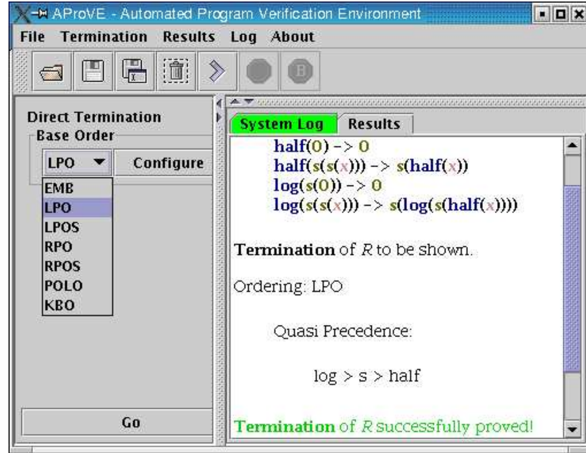
The system AProVE (Automated Program Verification Environment) offers a variety of techniques for automated termination proofs of TRSs: First, it provides efficient implementations of classical simplification orders to prove termination "directly" (*recursive path orders* possibly with status [6, 19], *Knuth-Bendix orders* [20], and *polynomial orders* [22]), cf. Sect. 2. To increase the power of automated termination proofs, we implemented the *dependency pair* technique [2, 13] in AProVE which allows the application of classical orders to examples where automated termination analysis would fail otherwise (Sect. 3). In contrast to most other implementations, we integrated numerous refinements such as *narrowing*, *rewriting*, and *instantiation* of dependency pairs [2, 12, 14, 15], recent improvements to reduce the constraints generated by the dependency pair technique [14, 15, 28], etc. Therefore, AProVE succeeds on many examples where all other automated termination provers fail. Thus, the principles used in AProVE's implementation may also be very helpful for other tools based on dependency pairs (Arts [1], CiME [5], TTT [18]) or on other related approaches for termination of TRSs (Termptation [4], Cariboo [10]). Apart from direct termination proofs and dependency pairs, as a third termination technique, AProVE offers the *size-change principle* [23] and it is also possible to combine this principle with dependency pairs [27] (Sect. 4). The tool is written in Java and proofs can be performed both in a fully automated or in an interactive mode via a graphical user interface. The modular design of AProVE's implementation is explained in Sect. 5. In Sect. 6 we show how to run the system and compare AProVE with related tools.

2 Direct Termination Proofs

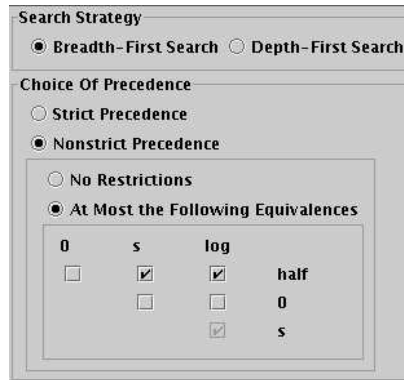
This section describes the base orders of AProVE which can be used for direct ter-

* *Proceedings of the 15th Int. Conference on Rewriting Techniques and Applications (RTA-04)*, Aachen, Germany, LNCS, Springer-Verlag, 2004.

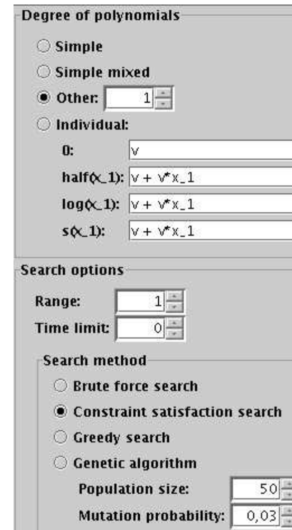
mination proofs, but also for proofs with constraint generation techniques like dependency pairs or the size-change principle.



In direct termination proofs, the system tries to find a reduction order where all rules are decreasing. The following *path orders* are available: the embedding order (EMB), the lexicographic path order (LPO, [19]), the LPO with status which compares subterms lexicographically w.r.t. arbitrary permutations (LPOS), the recursive path order comparing subterms as multisets (RPO, [6]), and the RPO with status which combines LPOS and RPO (RPOS).



Path orders may be parameterized by a precedence on function symbols and a status which determines how arguments of function symbols are compared. To explore the search space for these parameters, the system leaves them as unspecified (or "*minimal*") as possible. The user can decide between depth-first or breadth-first search and one can configure path orders by deciding whether different function symbols may be equivalent w.r.t. the precedence ("Nonstrict Precedence"). It is also possible to restrict potential equivalences to certain pairs of function symbols.

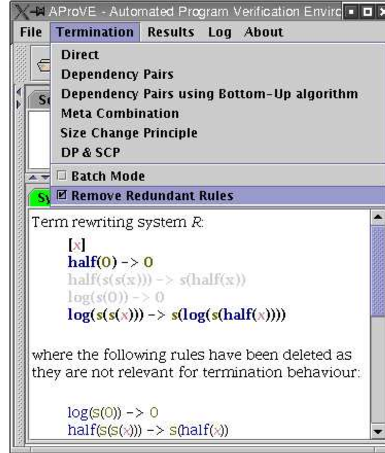


AProVE also offers *Knuth-Bendix orders* (KBO, [20]) using the polynomial-time algorithm of [21] and the technique of [9] to compute the degenerate subsystem of homogeneous linear inequalities.

The last class of orders in AProVE are *polynomial orders* (POLO, [22]) where every function symbol is associated with a polynomial with natural coefficients. The user can specify the degree of the polynomials and the range of the coefficients. One can also provide individual polynomials for some function symbols manually. To prove termination, AProVE generates a set of polynomial inequalities stating that left-hand sides of rules should be greater than the

corresponding right-hand sides. By the method of partial derivation [11, 22], these inequalities are transformed into inequalities only containing coefficients, but no variables anymore. Finally, a search algorithm determines suitable coefficients that satisfy the resulting inequalities. The user can choose between brute force search, greedy search, a genetic algorithm, and a constraint-based method based on interval arithmetic, which is the preferred one in most examples.

To improve power and efficiency of automated termination proofs, one can apply a pre-processing step to remove rules from the TRS that do not influence the termination behavior. When selecting "`Remove Redundant Rules`", AProVE tries to find a monotonic order $\succ$ such that the rules of the TRS $\mathcal{R}$ are at least weakly decreasing (i.e., at least $l \succsim r$ for all $l \to r \in \mathcal{R}$). Then rules which are strictly decreasing are removed, i.e., it suffices to prove termination of $\mathcal{R} \setminus \{l \to r \mid l \succ r\}$. This extends existing related approaches to remove rules [16, 22, 30].



For this pre-processing, we use linear polynomial interpretations with coefficients from $\{0, 1\}$. (In the screenshot above, we mapped $\mathsf{s}(x)$ to $x + 1$ and half and log to the identity.) AProVE's algorithm for polynomial orders solves constraints where some inequalities are strictly decreasing and all others are weakly decreasing in just one search attempt without backtracking [15]. So removal of rules can be done very efficiently and it is repeated until no rule can be removed anymore.

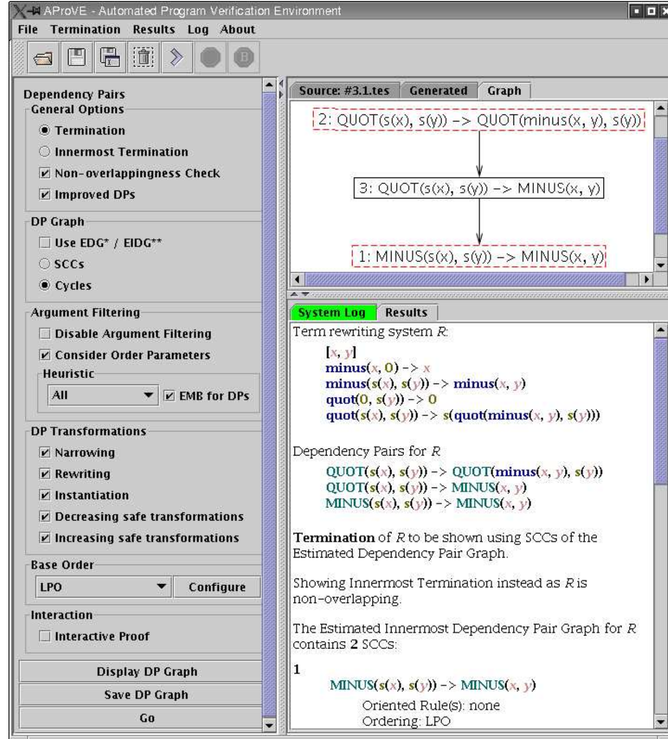
3 Termination Proofs with Dependency Pairs

The *dependency pair* approach [2, 13] increases the power of automated termination analysis significantly. The root symbols of left-hand sides of rules are called *defined* and all other symbols are *constructors*. For each defined symbol f we introduce a fresh *tuple symbol* F. Then for each rule $f(s_1, ..., s_n) \to r$ and each subterm $g(t_1, ..., t_m)$ of r with defined root g, we build a dependency pair $F(s_1, ..., s_n) \to G(t_1, ..., t_m)$. To prove termination one has to find a weakly monotonic order $\succ$ such that $s \succ t$ for all dependency pairs $s \to t$ and $l \succsim r$ for all rules $l \to r$. For innermost termination, $l \succsim r$ is only required for the usable rules of defined symbols in right-hand sides of dependency pairs. The *usable rules* for f are the f-rules together with the usable rules for all defined symbols in right-hand sides of f-rules. Moreover, if $\succsim$ is $\mathcal{C}_\varepsilon$-compatible (which holds for all orders in Sect. 2), then even for termination one only has to require $l \succsim r$ for the usable rules [28].

General Options and Base Order

In AProVE, one can select whether to use dependency pairs for termination or for

innermost termination proofs. The system also checks if a TRS is non-overlapping (then innermost termination implies termination). AProVE contains recent improvements which combine different modularity criteria and reduce the set of usable rules [14, 28].[1] They can be switched off for experimental purposes. To search for orders $\succ$, one can select any base order from Sect. 2.



Argument Filter

However, most of these orders are *strongly* monotonic, while the dependency pair approach only requires *weak* monotonicity. (For polynomial orders, a weakly monotonic variant is obtained by permitting the coefficient 0. But `LPO(S)`, `RPO(S)`, and `KBO` are always strongly monotonic.) Thus, before searching for an order, some of the arguments of the function symbols in the constraints can be eliminated by an *argument filtering* π [2]. For example, a binary function symbol f can be turned into a unary symbol by eliminating f's first argument. Then π replaces all terms $f(t_1, t_2)$ in the constraints by $f(t_2)$. Hence, we can obtain a weakly monotonic order $\succ_\pi$ from a strongly monotonic order $\succ$ and an argument filtering π by defining $s \succ_\pi t$ iff $\pi(s) \succ \pi(t)$. Moreover, we developed an improvement by first applying the argument filtering and determining the usable rules afterwards [14, 28]. The advantage is that the argument filtering may eliminate some symbols f from the right-hand sides of dependency pairs and rules. Then, one does not have to require $l \succsim_\pi r$ for the f-rules anymore. For this improvement, one has to select "`Improved DPs`" in the `General Options`.

As there are exponentially many argument filterings, this search space must be explored efficiently. AProVE uses a depth-first algorithm [14] which treats the constraints one after another. We start with the set of argument filterings possibly satisfying the first constraint. Here we use the idea of [17] to keep argument filterings as "undefined" as possible. Then this set is reduced further to

[1] Currently, the results of [28] are only available in AProVE 1.1-beta, which does not yet contain all options of AProVE 1.0. AProVE 1.1 will combine their features.

those filterings which possibly satisfy the second constraint as well. This procedure is repeated until all constraints are investigated. By inspecting constraints in a suitable order (instead of treating them separately as in [17]), already after the first constraint the set of possible argument filterings is rather small. Thus, one only inspects a fraction of all potential argument filterings. To use our refinement of filtering before computing usable rules, we also developed an algorithm to determine suitable filterings in this improved approach automatically, which is non-trivial since the filtering determines the resulting constraints.

One can also combine the search for the argument filtering with the search for the base order by choosing the option "`Consider Order Parameters`". Then the system also stores the corresponding parameters of the order for each possible argument filtering (e.g., a minimal set of precedences and stati, cf. Sect. 2).

Heuristics

To improve performance, one can use heuristics to restrict the set of possible argument filterings [14]. The most successful heuristic "`Type`" only regards argument filterings where for every symbol f, either no argument position is eliminated or all non-eliminated argument positions are of the same type. Here, we use a monomorphic type inference algorithm to transform a TRS into a sorted TRS (where in every rule $l \to r$, l and r must be well-typed terms of the same type).

When selecting the heuristic "`EMB for DPs`", only the very simple embedding order is used for orienting constraints like $s \succ_\pi t$ which come from dependency pairs $s \to t$. Only for constraints $l \succsim_\pi r$ from rules $l \to r$, one may apply more complicated orders like `LPO`, `RPO(S)`, etc. Since our depth-first algorithm to determine argument filterings starts with the dependency pairs, this reduces the search space significantly without compromising power very much.

This depth-first algorithm uses a top-down approach where constraints from f-rules are considered before g-rules, if f calls g. As an alternative heuristic, we also offer a "`Bottom-Up algorithm`" which starts with determining an argument filtering for constructors and then moves upwards through the recursion hierarchy where g is treated before f, if f calls g. To obtain an efficient technique, here the system only determines one single argument filtering at each choice point, even if several ones are possible and it does not perform any backtracking. This algorithm reduces the search space enormously, but is also restricts the power, since the proof can fail if one selects the "wrong" argument filtering at some point. Thus, this heuristic is suitable as a fast pre-processing step and if it fails, one should still apply the full dependency pair approach afterwards, cf. Sect. 5.

DP Graph

For TRSs with many rules, (innermost) termination proofs should be performed in a modular way. To this end, one constructs an estimated (innermost) dependency graph and regards its cycles separately [2, 13]. One can select between standard [2] and more powerful recent estimations (`EDG`* `/ EIDG`**) [15, 17].

For each cycle, only one dependency pair must be strictly decreasing and the others just have to be weakly decreasing. As shown in [17], one should not com-

pute all cycles, but only maximal cycles (*strongly connected components (SCCs)*). The reason is that the chosen argument filtering and base order may make several dependency pairs in an SCC strictly decreasing. In that case, subcycles of the SCC containing such a strictly decreasing dependency pair do not have to be considered anymore. So after solving the constraints for the initial SCCs, all strictly decreasing dependency pairs are removed and one now builds SCCs from the remaining dependency pairs, etc. This algorithm is chosen by selecting "`Cycles`". The algorithm "`SCCs`" requires a strict decrease for all dependency pairs in an SCC and is only intended for experimental purposes.

In order to benefit from all existing refinements on modularity of dependency pairs, we developed and implemented an improved technique which permits the combination of recent results on modularity of $\mathcal{C}_\varepsilon$-terminating TRSs [29] with arbitrary estimations of dependency graphs, cf. [14, 28]. This improvement is only available if one selects "`Improved DPs`" in the `General Options`.

DP Transformations

To increase power, a dependency pair can be transformed into several new pairs by *narrowing*, *rewriting*, and *instantiation* [2, 12, 14, 15]. In contrast to [12, 14], AProVE can instantiate dependency pairs both w.r.t. the pairs before and behind it in chains (the latter is called *forward* instantiation) [15]. The user can select which of these transformations should be applicable. Usually, all transformations should be enabled, since they are often crucial for the success of the proof and they can never "harm": if the termination proof succeeds without transformations, then it also succeeds when performing transformations [15], but not vice versa. However, the problem is when to use these transformations, since they may be applicable infinitely often. Moreover, transformations may increase runtime by producing a large number of similar constraints. AProVE performs transformations in "safe" cases where their application is guaranteed to terminate [14]. We distinguish between *increasing* and *decreasing* safe transformations. Decreasing transformations delete dependency pairs or SCCs and therefore, they do not have a negative impact on the efficiency. The user can disable both kinds of safe transformations. If turned on, decreasing transformations are applied before trying to solve the constraints for an SCC. Increasing transformations are only used a limited number of times when a proof attempt fails, and then the proof is re-attempted again.

Interaction

In addition to the fully

automated mode, (innermost) termination proofs with dependency pairs can be performed interactively. Here, the user can specify which transformation steps should be performed and for any cycle or SCC, one can determine (parts of) the argument filtering, the base order, and the dependency pair which should be strictly decreasing. The constraints resulting from such selections are immediately displayed, such that interactive proofs are supported in a very comfortable way. This mode is intended for the development of new heuristics and for machine-assisted proofs of particularly challenging examples.

4 Termination Proofs with the Size-Change Principle

A new *size-change principle* for termination of functional programs was presented in [23] and extended to TRSs in [27]. A similar principle is also known for logic programs [8]. AProVE offers the technique of [27, Thm. 11] for size-change termination of TRSs using the embedding order as underlying base order.[2]

AProVE also contains the combination of the size-change principle with dependency pairs from [27], which often succeeds with much simpler argument filterings and base orders than the pure dependency pair approach. Again, each SCC of the estimated (innermost) dependency graph is treated separately. In case of failure for some SCC, the dependency pairs are transformed by narrowing, rewriting, or instantiation and the proof attempt is re-started. If the user has selected the "`hybrid`" algorithm, then the pure dependency pair method is tried as soon as the limits for the transformations are reached. Thus, then the combined method is used as a fast technique which is checked first for every SCC and only if it fails, one uses the ordinary dependency pair approach on this SCC.

5 Design of AProVE's Implementation

All techniques of the previous two sections are *SCC-processors* which transform one SCC into a set of new SCCs: The dependency pair approach takes an SCC and if the constraints for this SCC can be solved using some base order, it deletes the strictly decreasing dependency pairs and returns the SCCs of the remaining subgraph. The DP transformations also produce a set of new SCCs out of a given one. Finally, the combination of dependency pairs with the size-change principle processes the SCCs of the estimated (innermost) dependency graph one by one, too. Therefore, all these techniques are implemented as modules which take one SCC as input and return a set of SCCs. So AProVE uses the following main algorithm, where one may choose different SCC-processors in Step 4 (b).

```
1. Remove redundant rules of the TRS which do not influence termination.
2. Check whether the TRS is non-overlapping. Then it is sufficient to
   prove innermost termination instead of termination.
3. Compute initial SCCs of the estimated (innermost) dependency graph.
```

[2] As shown in [27], only very restricted base orders are sound in connection with the size-change principle. In addition to the results in [27], the full embedding order may be used, where $f(\ldots, x_i, \ldots) \succ x_i$ also holds for defined function symbols f.

4. `While there are SCCs left and there is no failure:`
   (a) `Remove one SCC` $\mathcal{P}$ `from the set of SCCs.`
   (b) `Choose an SCC-processor.`
   (c) `Transform` $\mathcal{P}$ `with the chosen SCC-processor.`
   (d) `Add the resulting new set of SCCs to the remaining SCCs.`

Due to this modular structure, procedures which combine different termination techniques can easily be implemented in AProVE. One just has to configure which SCC-processors should be taken in Step 4 (b). It is advantageous if one first tries to use fast SCC-processors which benefit from successful heuristics. In this way, SCCs that are easy to handle can be treated efficiently. Only for SCCs where these fast SCC-processors fail, one should use slower but more powerful SCC-processors afterwards. Examples for such termination procedures offered in AProVE are the `hybrid` algorithm described in Sect. 4 or the following "`Meta Combination`" algorithm. This algorithm is particularly useful if one does not want to get involved with the details of termination proving, but wants to use AProVE in a "black box"-mode. In Step 4 (b), it always takes the first processor from the following list that is applicable (i.e., that can transform the SCC $\mathcal{P}$ into a new set of SCCs different from $\mathcal{P}$). Here, we use linear polynomial interpretations with coefficients from $\{0, 1\}$ and LPOs with "`Nonstrict Precedence`".

- `Decreasing safe transformations`
- "`DPs using Bottom-Up algorithm`" `with POLO and LPO as base orders`
- `Dependency pairs with the heuristic` "`EMB for DPs`" `and LPO`
- `Full dependency pair approach with POLO as base order`
- `Increasing safe transformations`

6 Running AProVE and Comparison with Other Tools

AProVE accepts four input languages: logic and (first-order) functional programs, conditional and unconditional TRSs. Functional and logic programs are translated into conditional TRSs and conditional TRSs are transformed further into unconditional TRSs [12, 24]. For logic programs, these transformations correspond to the approach of the termination prover TALP [25].

The results of the termination proof are displayed in `html`-format and can be stored in `html`- or LaTeX-format. Moreover, a "`System Log`" describes all (possibly failed) proof attempts. Any termination proof attempt may be interrupted by a stop-button. Instead of running the system on only one TRS or program, one can also run it on collections and directories of examples in a "`Batch Mode`".

Compared with other recent automated termination provers for TRSs (Arts [1], Cariboo [10], CiME [5], Termptation [4], TTT [18]), AProVE is the only system incorporating improvements like automated dependency pair transformations, applying argument filterings before usable rules, and combining modularity results based on $\mathcal{C}_\varepsilon$-termination with recent dependency graph estimations. Moreover, it offers more base orders than any other system, it can also handle conditional TRSs, and integrates the size-change principle. Finally, AProVE's design permits the combination of powerful heuristics and different termination techniques as in the "`Meta Combination`" algorithm of Sect. 5. In addition, the

system has a graphical user interface and a comfortable interactive component.

The next version of AProVE will also feature AC-rewriting and we try to improve its performance on string rewrite systems and logic programs. Our future work is also concerned with extensions to handle imperative programs, higher-order functional programs, and context-sensitive rewriting. Moreover, we plan to add a component to detect programs and systems that are *not* terminating.

Due to the numerous improvements developed and integrated in AProVE, it succeeded on more examples than any other system at the exhibition/competition of automated termination provers at the International Termination Workshop 2003. These results are confirmed by the following experiments, where we give an empirical comparison of AProVE 1.0 (using the "`Meta Combination`" algorithm) with the only two other tools currently available on the web (CiME and Termptation). The tools were tested on the collections of [3, 7, 26] (130 TRSs for termination, 151 TRSs for innermost termination). To show that the techniques described in [18] are a substantial restriction, in the last row we ran AProVE in a mode where we switched off all improvements and only used the methods available in [18]. Since [18] contains several base orders and argument filtering heuristics, we took the ones which gave the best overall result on this collection.

	Termination		Innermost Term.	
System	Power	Time	Power	Time
AProVE	95.4 %	26.2 s	98.0 %	34.3 s
CiME	71.5 %	660.7 s	—	—
Termptation	65.4 %	521.8 s	72.8 %	681.7 s
AProVE with techniques of [18]	52.3 %	679.1 s	—	—

The "Power" column contains the percentage of those examples in the collection where the proof attempt was successful. The "Time" column gives the overall time for running the system on all examples of the collection (also on the ones where the proof attempt failed). For each example we used a time-out of 60 seconds on a Pentium IV with 2.4 GHz and 1 GB memory. For more details on the above experiments and to download AProVE, the reader is referred to `http://www-i2.informatik.rwth-aachen.de/AProVE`.

References

1. T. Arts. System description: The dependency pair method. In *Proc. 11th RTA*, LNCS 1833, pages 261–264, 2000.
2. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
3. T. Arts and J. Giesl. A collection of examples for termination of term rewriting using dependency pairs. Technical Report AIB-2001-09[3], RWTH Aachen, 2001.
4. C. Borralleras, M. Ferreira, and A. Rubio. Complete monotonic semantic path orderings. In *Proc. 17th CADE*, LNAI 1831, pages 346–364, 2000.
5. E. Contejean, C. Marché, B. Monate, and X. Urbain. CiME. `http://cime.lri.fr`.

[3] Available from `http://aib.informatik.rwth-aachen.de`

6. N. Dershowitz. Termination of rewriting. *J. Symb. Comp.*, 3:69–116, 1987.
7. N. Dershowitz. 33 examples of termination. In *Proc. French Spring School of Theoretical Computer Science*, LNCS 909, pages 16–26, 1995.
8. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1,2):117–156, 2001.
9. J. Dick, J. Kalmus, and U. Martin. Automating the Knuth-Bendix ordering. *Acta Informatica*, 28:95–119, 1990.
10. O. Fissore, I. Gnaedig, and H. Kirchner. Cariboo: An induction based proof tool for termination with strategies. In *Proc. 4th PPDP*, pages 62–73. ACM, 2002.
11. J. Giesl. Generating polynomial orderings for termination proofs. In *Proc. 6th RTA*, LNCS 914, pages 426–431, 1995.
12. J. Giesl and T. Arts. Verification of Erlang processes by dependency pairs. *Appl. Algebra in Engineering, Communication and Computing*, 12(1,2):39–72, 2001.
13. J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.
14. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Improving dependency pairs. In *Proc. 10th LPAR*, LNAI 2850, pages 165–179, 2003.
15. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing dependency pairs. Technical Report AIB-2003-08[3], RWTH Aachen, Germany, 2003.
16. J. Giesl and H. Zantema. Liveness in rewriting. In *Proc. 14th RTA*, LNCS 2706, pages 321–336, 2003.
17. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. In *Proc. 19th CADE*, LNAI 2741, 2003.
18. N. Hirokawa and A. Middeldorp. Tsukuba termination tool. In *Proc. 14th RTA*, LNCS 2706, pages 311–320, 2003.
19. S. Kamin and J. J. Lévy. Two generalizations of the recursive path ordering. Unpublished Manuscript, University of Illinois, IL, USA, 1980.
20. D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Comp. Problems in Abstract Algebra*, pages 263–297. Pergamon, 1970.
21. K. Korovin and A. Voronkov. Verifying orientability of rewrite rules using the Knuth-Bendix order. In *Proc. 10th RTA*, LNCS 2051, pages 137–153, 2001.
22. D. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.
23. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proc. POPL '01*, pages 81–92, 2001.
24. E. Ohlebusch. Termination of logic programs: Transformational approaches revisited. *Appl. Algebra in Engineering, Comm. and Comp.*, 12(1,2):73–116, 2001.
25. E. Ohlebusch, C. Claves, and C. Marché. TALP: A tool for the termination analysis of logic programs. In *Proc. 11th RTA*, LNCS 1833, pages 270–273, 2000.
26. J. Steinbach. Automatic termination proofs with transformation orderings. In *Proc. 6th RTA*, LNCS 914, pages 11–25, 1995. Full version appeared as Technical Report SR-92-23, Universität Kaiserslautern, Germany.
27. R. Thiemann and J. Giesl. Size-change termination for term rewriting. In *Proc. 14th RTA*, LNCS 2706, pages 264–278, 2003.
28. R. Thiemann, J. Giesl, and P. Schneider-Kamp. Improved modular termination proofs using dependency pairs. In *Proc. 2nd IJCAR*, LNAI, 2004. To appear.
29. X. Urbain. Automated incremental termination proofs for hierarchically defined term rewriting systems. In *Proc. 1st IJCAR*, LNAI 2083, pages 485–498, 2001.
30. H. Zantema. TORPA: Termination of rewriting proved automatically. In *Proc. 15th RTA*, LNCS, 2004.