RWTH Aachen

Department of Computer Science
Technical Report

Dependency Triples for Improving Termination Analysis of Logic Programs with Cut

Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2010-12

RWTH Aachen · Department of Computer Science · Nov. 2010 (revised version)

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Dependency Triples for Improving Termination Analysis of Logic Programs with Cut*

Thomas Ströder[1], Peter Schneider-Kamp[2], and Jürgen Giesl[1]

[1] LuFG Informatik 2, RWTH Aachen University, Germany
{stroeder,giesl}@informatik.rwth-aachen.de
[2] IMADA, University of Southern Denmark, Denmark
petersk@imada.sdu.dk

Abstract. In very recent work, we introduced a non-termination preserving transformation from logic programs with cut to definite logic programs. While that approach allows us to prove termination of a large class of logic programs with cut automatically, in several cases the transformation results in a non-terminating definite logic program.
In this paper we extend the transformation such that logic programs with cut are no longer transformed into definite logic programs, but into dependency triple problems. By the implementation of our new method and extensive experiments, we empirically evaluate the practical benefit of our contributions.

1 Introduction

Automated termination analysis for logic programs has been widely studied, see, e.g., ([3–5, 13, 15, 16, 19]). Still, virtually all existing techniques only prove universal termination of *definite* logic programs, which do not use the *cut* "!".[3] But most realistic Prolog programs make use of the cut or related operators such as *negation as failure* ("\+") or *if then else* ("–> ... ; ..."), which can be expressed using cuts. In [18] we introduced a non-termination preserving automated transformation from logic programs with cut to definite logic programs. The transformation consists of two stages. The first stage is based on constructing a so-called *termination graph* from a given logic program with cut. The second stage is the generation of a definite logic program from this termination graph. In this paper, we improve the second stage of the transformation. Instead of generating a definite logic program from the termination graph, we now

* Supported by the DFG grant GI 274/5-3, the DFG Research Training Group 1298 (*AlgoSyn*), and the Danish Natural Science Research Council.

[3] An exception is [12], which presents a transformation of "safely-typed" logic programs to term rewrite systems. However, the resulting rewrite systems are quite complex and since there is no implementation of [12], it is unclear whether they can indeed be handled by existing termination tools from term rewriting. Moreover, [12] does not allow programs with arbitrary cuts (e.g., it does not operate on programs like the one in Ex. 1).

generate a dependency triple problem. The goal is to improve the power of the approach, i.e., to succeed also in many cases where the transformation of [18] yields a non-terminating definite logic program.

Dependency triples were introduced in [14] and improved further to the so-called *dependency triple framework* in [17]. The idea was to adapt the successful dependency pair framework [2, 8–10] from term rewriting to (definite) logic programming. This resulted in a completely modular method for termination analysis of logic programs which even allowed to combine "direct" and "transformational" methods within the proof of one and the same program. The experiments in [17] showed that this leads to the most powerful approach for automated termination analysis of definite logic programs so far. Our aim is to benefit from this work by providing an immediate translation from logic programs with cut (resp. from their termination graphs) to dependency triple problems.

This paper is structured as follows. After a short section on preliminaries, we recapitulate the construction of termination graphs in Sect. 3 and we demonstrate their transformation into definite logic programs in Sect. 4. Then, in Sect. 5 we illustrate the idea of dependency triples and introduce a novel transformation of termination graphs into dependency triple problems. We show that this new transformation has significant practical advantages in Sect. 6 and, finally, conclude in Sect. 7.

2 Preliminaries

See e.g. [1] for the basics of logic programming. We distinguish between individual cuts to make their scope explicit. Thus, we use a signature Σ containing all predicate and function symbols as well as all labeled versions of the cut operator $\{!_m/0 \mid m \in \mathbb{N}\}$. For simplicity we just consider terms $\mathcal{T}(\Sigma, \mathcal{V})$ and no atoms, i.e., we do not distinguish between predicate and function symbols.[4] A *query* is a sequence of terms from $\mathcal{T}(\Sigma, \mathcal{V})$. Let $\mathit{Goal}(\Sigma, \mathcal{V})$ be the set of all queries, where $\Box$ is the empty query. A *clause* is a pair $H \leftarrow B$ where the *head* H is from $\mathcal{T}(\Sigma, \mathcal{V})$ and the *body* B is a query. A *logic program* $\mathcal{P}$ (possibly with cut) is a finite sequence of clauses. $\mathit{Slice}(\mathcal{P}, t)$ are all clauses for t's predicate, i.e., $\mathit{Slice}(\mathcal{P}, p(t_1, ..., t_n)) = \{c \mid c = \text{"}p(s_1, ..., s_n) \leftarrow B\text{"} \in \mathcal{P}\}$.

A substitution σ is a function $\mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ and we often denote its application to a term t by $t\sigma$ instead of $\sigma(t)$. As usual, $\mathit{Dom}(\sigma) = \{X \mid X\sigma \neq X\}$ and $\mathit{Range}(\sigma) = \{X\sigma \mid X \in \mathit{Dom}(\sigma)\}$. The restriction of σ to $\mathcal{V}' \subseteq \mathcal{V}$ is $\sigma_{|\mathcal{V}'}(X) = \sigma(X)$ if $X \in \mathcal{V}'$, and $\sigma_{|\mathcal{V}'}(X) = X$ otherwise. A substitution σ is the *most general unifier* (mgu) of s and t iff $s\sigma = t\sigma$ and, whenever $s\gamma = t\gamma$ for some other unifier γ, there exists a δ such that $X\gamma = X\sigma\delta$ for all $X \in \mathcal{V}(s) \cup \mathcal{V}(t)$. If s and t have no mgu, we write $s \not\sim t$.

Let Q be a query $A_1, \ldots, A_m$, let c be a clause $H \leftarrow B_1, \ldots, B_k$. Then Q' is a *resolvent* of Q and c using θ (denoted $Q \vdash_{c,\theta} Q'$) if θ is the mgu of A_1 and H, and $Q' = (B_1, \ldots, B_k, A_2, \ldots, A_m)\theta$.

[4] To ease the presentation, in the paper we exclude terms with cuts $!_m$ as proper subterms.

A *derivation* of a program $\mathcal{P}$ and Q is a possibly infinite sequence $Q_0, Q_1, \ldots$ of queries with $Q_0 = Q$ where for all i, we have $Q_i \vdash_{c_{i+1},\theta_{i+1}} Q_{i+1}$ for some substitution θ_{i+1} and some fresh variant c_{i+1} of a clause from $\mathcal{P}$. For a derivation $Q_0, \ldots, Q_n$ as above, we also write $Q_0 \vdash^n_{\mathcal{P},\theta_1 \ldots \theta_n} Q_n$ or $Q_0 \vdash^n_{\mathcal{P}} Q_n$, and we also write $Q_i \vdash_{\mathcal{P}} Q_{i+1}$ for $Q_i \vdash_{c_{i+1},\theta_{i+1}} Q_{i+1}$. The query Q *terminates* for $\mathcal{P}$ if all derivations of $\mathcal{P}$ and Q are finite, i.e., if $\vdash_{\mathcal{P}}$ is terminating for Q. *Answer*$(\mathcal{P}, Q)$ is the set of all substitutions δ such that $Q \vdash^n_{\mathcal{P},\delta} \square$ for some $n \in \mathbb{N}$.

Finally, to denote the term resulting from replacing all occurrences of a function symbol f in a term t by another function symbol g, we write $t[f/g]$.

3 Termination Graphs

To illustrate the concepts and the contributions of this paper, we use the following leading example. While this example has been designed for this purpose, as demonstrated in Sect. 6, our contributions also have a considerable effect for termination analysis of "general" logic programs with cut.

Example 1. The following clauses define a (simplified) variant of the logic program `Stroeder09/giesl97.pl` from the Termination Problem Data Base [22] that is used in the annual international Termination Competition [21]. This example formulates a functional program from [7, 23] with nested recursion as a logic program. Here, the predicate p is used to compute the predecessor of a natural number while eq is used to unify two terms.

$$\mathsf{f}(\mathsf{0}, Y) \leftarrow !, \mathsf{eq}(Y, \mathsf{0}). \tag{1}$$

$$\mathsf{f}(X, Y) \leftarrow \mathsf{p}(X, P), \mathsf{f}(P, U), \mathsf{f}(U, Y). \tag{2}$$

$$\mathsf{p}(\mathsf{0}, \mathsf{0}). \tag{3}$$

$$\mathsf{p}(\mathsf{s}(X), X). \tag{4}$$

$$\mathsf{eq}(X, X). \tag{5}$$

Note that when ignoring cuts, this logic program is not terminating for the set of queries $\{\mathsf{f}(t_1, t_2) \mid t_1 \text{ is ground}\}$. To see this, consider the following derivation: $\mathsf{f}(\mathsf{0}, A) \vdash \mathsf{p}(\mathsf{0}, P), \mathsf{f}(P, U), \mathsf{f}(U, A) \vdash_{\{P/\mathsf{0}\}} \mathsf{f}(\mathsf{0}, U), \mathsf{f}(U, A) \vdash \mathsf{eq}(U, \mathsf{0}), \mathsf{f}(U, A) \vdash_{\{U/\mathsf{0}\}} \mathsf{f}(\mathsf{0}, A)$. Clearly, this leads to an infinite (looping) derivation.

Fig. 1 recapitulates the formulation of the operational semantics of logic programming with cut that we introduced in [18]. A formal proof on the correspondence of our inference rules to the semantics of the Prolog ISO standard [6] can be found in [20]. The formulation with our inference rules is particularly suitable for an extension to *classes* of queries in Fig. 2, and for synthesizing cut-free programs in Sect. 4 or dependency triples in Sect. 5. Our semantics is given by seven inference rules. They operate on *states* that do not just represent the current goal, but also the backtrack information that is needed to describe the effect of cuts. The backtrack information is given by a sequence of goals (separated by "|") which are optionally labeled by the program clause i that has to be applied to the goal next and by a number m that determines how cuts will

$$\frac{\Box \mid S}{S}\ (\textsc{Suc}) \qquad \frac{?_m \mid S}{S}\ (\textsc{Fail}) \qquad \frac{!_m, Q \mid S \mid ?_m \mid S'}{Q \mid ?_m \mid S'}\ (\textsc{Cut})^{\text{where } S \text{ contains no } ?_m} \qquad \frac{!_m, Q \mid S}{Q}\ (\textsc{Cut})^{\text{where } S \text{ contains no } ?_m}$$

$$\frac{t, Q \mid S}{(t, Q)^{i_1}_m \mid \ldots \mid (t, Q)^{i_k}_m \mid ?_m \mid S}\ (\textsc{Case})$$ where t is neither a cut nor a variable, m is greater than all previous marks, and $Slice(\mathcal{P}, t) = \{c_{i_1}, \ldots, c_{i_k}\}$ with $i_1 < \ldots < i_k$

$$\frac{(t, Q)^i_m \mid S}{B'_i\sigma, Q\sigma \mid S}\ (\textsc{Eval})^{\text{where } c_i = H_i \leftarrow B_i,\ mgu(t, H_i) = \sigma,\ B'_i = B_i[! \,/\, !_m].} \qquad \frac{(t, Q)^i_m \mid S}{S}\ (\textsc{Backtrack})^{\text{where } c_i = H_i \leftarrow B_i \text{ and } t \not\sim H_i.}$$

Fig. 1. Operational Semantics by Concrete Inference Rules

be labeled when evaluating this goal later on. Moreover, our states also contain explicit *marks* $?_m$ to mark the end of the scope of a cut $!_m$.

For the query $\mathsf{f}(0, A)$ in Ex. 1, we obtain the following derivation with the rules of Fig. 1: $\mathsf{f}(0, A) \vdash_{\textsc{Case}} \mathsf{f}(0, A)^1_1 \mid \mathsf{f}(0, A)^2_1 \mid ?_1 \vdash_{\textsc{Eval}} !_1, \mathsf{eq}(A, 0) \mid \mathsf{f}(0, A)^2_1 \mid ?_1 \vdash_{\textsc{Cut}} \mathsf{eq}(A, 0) \mid ?_1 \vdash_{\textsc{Case}} \mathsf{eq}(A, 0)^5_2 \mid ?_2 \mid ?_1 \vdash_{\textsc{Eval}} \Box \mid ?_2 \mid ?_1 \vdash_{\textsc{Suc}} ?_2 \mid ?_1 \vdash_{\textsc{Fail}} ?_1 \vdash_{\textsc{Fail}} \varepsilon$. Thus, when considering cuts, our logic program terminates for the query $\mathsf{f}(0, A)$, and, indeed, it terminates for all queries from the set $\{\mathsf{f}(t_1, t_2) \mid t_1 \text{ is ground}\}$. For further details on the intuition behind the inference rules, we refer to [18].

To show termination for infinite sets of queries (e.g., $\{\mathsf{f}(t_1, t_2) \mid t_1 \text{ is ground}\}$), we need to represent classes of queries by abstract states. To this end, in [18] we introduced *abstract terms* and a set $\mathcal{A}$ of *abstract variables*, where each $T \in \mathcal{A}$ represents a fixed but arbitrary term. $\mathcal{N}$ consists of all "ordinary" variables in logic programming. Then, as *abstract terms* we consider all terms from the set $\mathcal{T}(\Sigma, \mathcal{V})$ where $\mathcal{V} = \mathcal{N} \uplus \mathcal{A}$. *Concrete terms* are terms from $\mathcal{T}(\Sigma, \mathcal{N})$, i.e., terms containing no abstract variables. For any set $\mathcal{V}' \subseteq \mathcal{V}$, let $\mathcal{V}'(t)$ be the variables from $\mathcal{V}'$ occurring in the term t. To determine by which terms an abstract variable may be instantiated, we add a knowledge base $KB = (\mathcal{G}, \mathcal{U})$ to each state, where $\mathcal{G} \subseteq \mathcal{A}$ and $\mathcal{U} \subseteq \mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{T}(\Sigma, \mathcal{V})$. Instantiations γ that respect KB may only instantiate the variables in $\mathcal{G}$ by ground terms. And $(s, s') \in \mathcal{U}$ means that we are restricted to instantiations γ where $s\gamma \not\sim s'\gamma$, i.e., s and s' may not become unifiable when instantiating them with γ. We call a substitution γ that respects the information in KB a *concretization* w.r.t. KB.

Fig. 2 shows the abstract inference rules introduced in [18]. They work on classes of queries represented by abstract terms with a knowledge base. Except for Backtrack and Eval, the adaption of the concrete inference rules to corresponding abstract inference rules is straightforward.

For Backtrack and Eval we must consider that the set of queries represented by an abstract state may contain both queries where the concrete Eval rule and where the concrete Backtrack rule is applicable. Thus, the abstract Eval rule has two successors corresponding to these two cases. As abstract variables not known to represent ground terms may share variables, we have to replace all variables by fresh abstract variables in Eval's left successor state which corresponds to the application of the concrete Eval rule. For the backtrack

$$\frac{\Box \mid S; KB}{S; KB}\ (\textsc{Suc}) \qquad \frac{?_m \mid S; KB}{S; KB}\ (\textsc{Fail})$$

$$\frac{!_m, Q \mid S \mid ?_m \mid S'; KB}{Q \mid ?_m \mid S'; KB}\ (\textsc{Cut})\ \begin{array}{l}\text{where } S \\ \text{contains no} \\ ?_m\end{array} \qquad \frac{!_m, Q \mid S; KB}{Q; KB}\ (\textsc{Cut})\ \begin{array}{l}\text{where } S \\ \text{contains no} \\ ?_m\end{array}$$

$$\frac{t, Q \mid S; KB}{(t, Q)^{i_1}_m \mid \ldots \mid (t, Q)^{i_k}_m \mid ?_m \mid S; KB}\ (\textsc{Case})$$

where t is neither a cut nor a variable, m is greater than all previous marks, and $Slice(\mathcal{P}, t) = \{c_{i_1}, \ldots, c_{i_k}\}$ with $i_1 < \ldots < i_k$

$$\frac{(t, Q)^i_m \mid S; KB}{S; KB}\ (\textsc{Backtrack})$$

where $c_i = H_i \leftarrow B_i$ and there is no concretization γ w.r.t. KB such that $t\gamma \sim H_i$.

$$\frac{(t, Q)^i_m \mid S; (\mathcal{G}, \mathcal{U})}{B'_i\sigma, Q\sigma \mid S\sigma|_{\mathcal{G}'}; (\mathcal{G}', \mathcal{U}\sigma|_{\mathcal{G}'}) \qquad S; (\mathcal{G}, \mathcal{U} \cup \{(t, H_i)\})}\ (\textsc{Eval})$$

where $c_i = H_i \leftarrow B_i$ and $mgu(t, H_i) = \sigma$. W.l.o.g., for all $X \in \mathcal{V}$, $\mathcal{V}(\sigma(X))$ only contains fresh abstract variables not occurring in $t, Q, S, \mathcal{G}$, or $\mathcal{U}$. Moreover, $\mathcal{G}' = \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ and $B'_i = B_i[!/!_m]$.

$$\frac{S; (\mathcal{G}, \mathcal{U})}{S'; (\mathcal{G}', \mathcal{U}')}\ (\textsc{Instance})$$

if there is a μ such that $S = S'\mu$, $\mu|_{\mathcal{N}}$ is a variable renaming, $\mathcal{V}(\mathcal{T}\mu) \subseteq \mathcal{G}$ for all $\mathcal{T} \in \mathcal{G}'$, and $\mathcal{U}'\mu \subseteq \mathcal{U}$.

$$\frac{S \mid S'; KB}{S; KB \qquad S'; KB}\ (\textsc{Parallel})$$

if $AC(S) \cap AM(S') = \varnothing$

The *active cuts* $AC(S)$ are all m where $!_m$ is in S or $(t, q)^i_m$ is in S and c_i's body has a cut. The *active marks* $AM(S)$ are all m where $S = S' \mid ?_m \mid S''$ and $S' \neq \varepsilon \neq S''$.

$$\frac{t, Q; (\mathcal{G}, \mathcal{U})}{t; (\mathcal{G}, \mathcal{U}) \qquad Q\mu; (\mathcal{G}', \mathcal{U}\mu)}\ (\textsc{Split})$$

where μ replaces all (abstract and non-abstract) variables from $\mathcal{V} \setminus \mathcal{G}$ by fresh abstract variables and $\mathcal{G}' = \mathcal{G} \cup ApproxGnd(t, \mu)$.

Here, we assume that we have a *groundness analysis* function $Ground_p : \Sigma \times 2^{\mathbb{N}} \to 2^{\mathbb{N}}$, see, e.g., [11]. Then we have $ApproxGnd(p(t_1, \ldots, t_n), \mu) = \{\mathcal{A}(t_j\mu) \mid j \in Ground_p(p, \{i \mid \mathcal{V}(t_i) \sqsubseteq \mathcal{G}\})\}$.

Fig. 2. Abstract Inference Rules

possibilities and the knowledge base, however, we may only use an answer substitution $\sigma|_{\mathcal{G}}$ restricted to abstract variables known to represent ground terms. The reason is that, due to backtracking, any substitutions of non-abstract variables may become canceled. For the abstract variables $T \in \mathcal{G}$, their instantiation with the answer substitution of the abstract Eval rule corresponds to a case analysis over the shape of the ground terms that T is representing. Thus, in case of a successful unification we know that these terms must have a certain shape and we can keep this information also after backtracking. Fig. 3 shows how the rules of Fig. 2 can be applied to the initial state $\mathsf{f}(T_1, T_2)$ with the knowledge base $(\{T_1\}, \varnothing)$, which represents the set of queries $\{\mathsf{f}(t_1, t_2) \mid t_1 \text{ is ground}\}$. In Fig. 3 we applied the Eval rule to Node B, for example. Its left successor corresponds to the case where T_1 represents the ground term 0 and, thus, the goal $\mathsf{f}(T_1, T_2)$ unifies with the head of the first clause of the program. Here we can replace all occurrences of T_1 by 0, as (due to $T_1 \in \mathcal{G}$) 0 is the term represented by T_1. In contrast, as $T_2 \notin \mathcal{G}$, the replacement of T_2 with the fresh variable T_3 is not performed in the second backtracking goal $\mathsf{f}(0, T_2)^2_1$. The right successor of Node B corresponds to all cases where the unification with the head of Clause (1) fails.

Fig. 3. Termination Graph for Ex. 1.

While the abstract Eval rule is already sufficient for a sound simulation of all concrete derivations, the abstract Backtrack rule is virtually always needed for a successful termination analysis, since otherwise, the application of the abstract inference rules would almost always yield an infinite tree. To apply the abstract Backtrack rule to an abstract state, we have to ensure that this state does not represent any queries where the concrete Eval rule would be applicable. In Fig. 3 we applied the Backtrack rule to Node C, for example. This is crucial for the success of the termination proof. As we know from the application of the Eval rule to Node B, we are in the case where the first argument T_4 does not unify with 0. Hence, Clause (3) is not applicable to the first goal $\mathsf{p}(T_4, P)$ and

thus, we can safely apply the BACKTRACK rule to C.

With the first seven rules, we would obtain infinite trees for non-terminating programs. Even for terminating programs we may obtain infinite trees as there is no bound on the size of the terms represented by abstract variables. For a finite analysis we have to refer back to already existing states. This is done by the INSTANCE rule. The intuition for this rule is that we can refer back to a state representing a superset of queries compared to the current state. This can be ensured by finding a matcher μ which matches the more general state to the more specific state. This rule can also be used to generalize states instead of referring back to existing states. This is needed in case of "repeatedly growing" terms which would otherwise never lead to a state where we can find an already existing instance. Considering our example graph in Fig. 3, we applied the INSTANCE rule to refer Node F back to Node A with the matching substitution $\{T_1/T_6, T_2/U\}$.

Still, this is not enough to always obtain a finite termination graph. On the one hand, the evaluation of a program may yield growing backtracking sequences which never lead to a state with an already existing instance. On the other hand, the number of terms in a query may also grow and cause the same problem. For the first situation we need the PARALLEL rule which can separate the backtracking sequence into two states. The second problem is solved by the SPLIT rule which splits off the first term of a single query. Both rules may lose precision, but are often needed for a finite analysis. To reduce the loss of precision, we approximate the answer substitutions of evaluations for the first successors of SPLIT nodes using a groundness analysis. This analysis determines whether some variables will be instantiated by ground terms in every successful derivation of SPLIT's left child. Then in the right child, these variables can be added to $\mathcal{G}$ and all other variables are replaced by fresh abstract variables (this is necessary due to sharing). In Fig. 3 we applied the PARALLEL rule to Node G. In this way, we created its child H which is an instance of the earlier Node A. The SPLIT rule is used to separate the goal $f(T_6, U)$ from the remainder of the state in Node E. The resulting child F is again an instance of A. For E's second child G, the groundness analysis found out that the variable U in the goal $f(U, T_5)$ must be instantiated with a ground term T_7 during the evaluation of node F. Therefore, T_7 is added to the set $\mathcal{G}$ in Node G. This groundness information is important for the success of the termination proof. Otherwise, the state G would also represent non-terminating queries and hence, the termination proof would fail.

Using these rules, for Ex. 1 we obtain the termination graph depicted in Fig. 3. A *termination graph* is a finite graph where no rule is applicable to its leaves and where there is no cycle which only uses the INSTANCE rule. Note that by applying an adequate strategy, we can obtain a termination graph for *any* logic program automatically [18, Thm. 2]. To ease presentation, in the graph of Fig. 3, we always removed those abstract variables from the knowledge base that do not occur in any goal of the respective state. A termination graph without leaves that start with variables is called *proper*. (Our termination proof fails if the graph contains leaves starting with abstract variables, since they stand for any possible query.) Again, we refer to [18] for further details and explanations.

4 Transformation into Definite Logic Programs

We now explain the second stage of the transformation from [18], i.e., the transformation of termination graphs into definite logic programs. For more details and formal definitions see [18].

Termination graphs have the property that each derivation of the original program corresponds to a path through the termination graph. Thus, infinite derivations of the original program correspond to an infinite traversal of cycles in the graph. The basic idea of the transformation is to generate (cut-free) clauses for each cycle in the graph. Then termination of the (definite) logic program consisting of these clauses implies termination of the original program. To simulate the traversal of cycles, we generate clauses for paths starting at the child of an INSTANCE or SPLIT node or at the root node and ending in a SUC or INSTANCE node or in a left child of an INSTANCE or SPLIT node while not traversing other INSTANCE nodes or left children of INSTANCE or SPLIT nodes. The formal definition of paths for which we generate clauses is given below. Here, for a termination graph G, let INSTANCE(G) denote all nodes of G to which the rule INSTANCE has been applied (i.e., F and H). The sets SPLIT(G) and SUC(G) are defined analogously. For any node n, let $Succ(i, n)$ denote the i-th child of n.

Definition 1 (Clause Path [18]). *A path $\pi = n_1 \dots n_k$ in G is a* clause path *iff $k > 1$ and*

- $n_1 \in Succ(1, \text{INSTANCE}(G) \cup \text{SPLIT}(G))$ *or n_1 is the root of G,*
- $n_k \in \text{SUC}(G) \cup \text{INSTANCE}(G) \cup Succ(1, \text{INSTANCE}(G) \cup \text{SPLIT}(G))$,
- *for all $1 \leq j < k$, we have $n_j \notin \text{INSTANCE}(G)$,*[5] *and*
- *for all $1 < j < k$, we have $n_j \notin Succ(1, \text{INSTANCE}(G) \cup \text{SPLIT}(G))$.*

In the graph of Fig. 3 we find two clause paths ending in INSTANCE nodes. One path is from the root node A to the INSTANCE node F and one from the root node A to the INSTANCE node H. We introduce a fresh predicate symbol p_n for each node n, where these predicates have all distinct variables occurring in the node n as arguments.

For an INSTANCE node, however, we use the same predicate as for its child while applying the matching substitution used for the instantiation. Hence, for the nodes A, F, H in Fig. 3, we obtain the terms $\mathsf{p}_\text{A}(T_1, T_2)$, $\mathsf{p}_\text{A}(T_6, U)$, and $\mathsf{p}_\text{A}(T_7, T_8)$. To generate clauses for every clause path, we have to consider the substitutions along the paths and successively apply them to the heads of the new clauses. Thus, for the clause path from A to F, we obtain the clause $\mathsf{p}_\text{A}(\mathsf{s}(T_6), T_5) \leftarrow \mathsf{p}_\text{A}(T_6, U)$.

However, for cycles traversing right children of SPLIT nodes, the newly generated clause should contain an additional intermediate body atom. This is due to the fact that the derivation along such a path is only possible if the goal corresponding to the left child of the respective SPLIT node is successfully evaluated

[5] Note that $n_k \in Succ(1, \text{INSTANCE}(G))$ is possible although $n_{k-1} \notin \text{INSTANCE}(G)$, since n_k may have more than one parent node in G.

first. Hence, we obtain the clause $\mathsf{p_A}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{p_A}(T_6, T_7), \mathsf{p_A}(T_7, T_8)$ for the path from A to H. To capture the evaluation of left children of SPLIT nodes, we also generate clauses corresponding to evaluations of left SPLIT children, i.e., paths in the graph from such nodes to SUC nodes, possibly traversing cycles first. Thus, the path from A to the only SUC node D is also a clause path. To transform it into a new clause, we have to apply the substitutions between the respective SPLIT node and the end of the path. For SUC nodes, we do not introduce new predicates. Hence, we obtain the fact $\mathsf{p_A}(0, 0)$ for the path from A to D. Thus, the resulting definite logic program for the termination graph from Fig. 3 is the following. Note that here, T_5, T_6, T_7, T_8 are considered as normal variables.

$$\mathsf{p_A}(0, 0). \tag{6}$$

$$\mathsf{p_A}(\mathsf{s}(T_6), T_5) \leftarrow \mathsf{p_A}(T_6, U). \tag{7}$$

$$\mathsf{p_A}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{p_A}(T_6, T_7), \mathsf{p_A}(T_7, T_8). \tag{8}$$

Below we give the formal definition for the clauses and queries generated for clause paths. To ease the presentation, we assume that for any path π, we do not traverse a BACKTRACK, FAIL, or SUC node or the right successor of an EVAL node after traversing the left successor of an EVAL node. The more general case can be found in [18] and also in the extended definitions and proofs in the appendix.

Definition 2 (Logic Programs from Termination Graph [18]). *Let G be a termination graph whose root n is $(f(T_1, \ldots, T_m), (\{T_{i_1}, \ldots, T_{i_k}\}, \emptyset))$. We define $\mathcal{P}_G = \bigcup_{\pi \text{ clause path in } G} Clause(\pi)$ and $\mathcal{Q}_G = \{\mathsf{p_n}(t_1, \ldots, t_m) \mid t_{i_1}, \ldots, t_{i_k} \text{ are ground}\}$. For a path $\pi = n_1 \ldots n_k$, let $Clause(\pi) = Ren(n_1)\sigma_\pi \leftarrow I_\pi, Ren(n_k)$. For $n \in \text{SUC}(G)$, $Ren(n)$ is $\Box$ and for $n \in \text{INSTANCE}(G)$, it is $Ren(Succ(1, n))\mu$ where μ is the substitution associated with the INSTANCE node n. Otherwise, $Ren(n)$ is $\mathsf{p_n}(\mathcal{V}(n))$ where $\mathcal{V}(S; KB) = \mathcal{V}(S)$.*

Finally, σ_π and I_π are defined as follows. Here for a path $\pi = n_1 \ldots n_j$, the substitutions μ and σ are the labels on the outgoing edge of $n_{j-1} \in \text{SPLIT}(G)$ and $n_{j-1} \in \text{EVAL}(G)$, respectively.

$$\sigma_{n_1 \ldots n_j} = \begin{cases} id & \text{if } j = 1 \\ \sigma_{n_1 \ldots n_{j-1}}\, \mu & \text{if } n_{j-1} \in \text{SPLIT}(G),\ n_j = Succ(2, n_{j-1}) \\ \sigma_{n_1 \ldots n_{j-1}}\, \sigma & \text{if } n_{j-1} \in \text{EVAL}(G),\ n_j = Succ(1, n_{j-1}) \\ \sigma_{n_1 \ldots n_{j-1}} & \text{otherwise} \end{cases}$$

$$I_{n_j \ldots n_k} = \begin{cases} \Box & \text{if } j = k \\ Ren(Succ(1, n_j))\sigma_{n_j \ldots n_k},\ I_{n_{j+1} \ldots n_k} & \text{if } n_j \in \text{SPLIT}(G), n_{j+1} = Succ(2, n_j) \\ I_{n_{j+1} \ldots n_k} & \text{otherwise} \end{cases}$$

Unfortunately, in our example, the generated program (6)-(8) is not (universally) terminating for all queries of the form $\mathsf{p_A}(t_1, t_2)$ where t_1 is a ground term. To see this, consider the query $\mathsf{p_A}(\mathsf{s}(\mathsf{s}(0)), Z)$. We obtain the following derivation.

$$\mathsf{p_A}(\mathsf{s}(\mathsf{s}(0)), Z) \vdash_{(8)} \mathsf{p_A}(\mathsf{s}(0), T_7), \mathsf{p_A}(T_7, Z) \vdash_{(7)} \mathsf{p_A}(0, U), \mathsf{p_A}(T_7, Z) \vdash_{(6)} \mathsf{p_A}(T_7, Z)$$

The last goal has infinitely many successful derivations. The reason why the transformation fails is that in the generated logic program, we cannot distinguish between the evaluation of intermediate goals and the traversal of cycles of the termination graph, since we only have one evaluation mechanism. We often encounter such problems when the original program has clauses whose body contains at least two atoms $q_1(\ldots), q_2(\ldots)$, where both predicates q_1 and q_2 have recursive clauses and where the call of q_2 depends on the result of q_1. This is a very natural situation occurring in many practical programs (cf. our experiments in Sect. 6). It is also the case in our example for the second clause (2) (here we have the special case where both q_1 and q_2 are equal).

5 Transformation into Dependency Triple Problems

To solve the problem illustrated in the last section, we modify the second stage of the transformation to construct dependency triple problems [17] instead of definite logic programs. The advantage of dependency triple problems is that they support two different kinds of evaluation which suit our needs to handle the evaluation of intermediate goals and the traversal of cycles differently.

The basic structure in the dependency triple framework is very similar to a clause in logic programming. Indeed, a *dependency triple* (DT) [14] is just a clause $H \leftarrow I, B$ where H and B are atoms and I is a sequence of atoms. Intuitively, such a DT states that a call that is an instance of H can be followed by a call that is an instance of B if the corresponding instance of I can be proven.

Here, a "derivation" is defined in terms of a chain. Let $\mathcal{D}$ be a set of DTs, $\mathcal{P}$ be the program under consideration, and $\mathcal{Q}$ be the class of queries to be analyzed.[6] A (possibly infinite) sequence $(H_0 \leftarrow I_0, B_0), (H_1 \leftarrow I_1, B_1), \ldots$ of variants from $\mathcal{D}$ is a $(\mathcal{D}, \mathcal{Q}, \mathcal{P})$-*chain* iff there are substitutions θ_i, σ_i and an $A \in \mathcal{Q}$ such that $\theta_0 = mgu(A, H_0)$ and for all i, we have $\sigma_i \in Answer(\mathcal{P}, I_i\theta_i)$ and $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$. Such a tuple $(\mathcal{D}, \mathcal{Q}, \mathcal{P})$ is called a *dependency triple problem* and it is *terminating* iff there is no infinite $(\mathcal{D}, \mathcal{Q}, \mathcal{P})$-chain.

As an example, consider the DT problem $(\mathcal{D}, \mathcal{Q}, \mathcal{P})$ with $\mathcal{D} = \{d_1\}$ where $d_1 = \mathsf{p}(\mathsf{s}(X), Y) \leftarrow \mathsf{eq}(X, Z), \mathsf{p}(Z, Y)$, $\mathcal{Q} = \{\mathsf{p}(t_1, t_2) \mid t_1 \text{ is ground}\}$, and $\mathcal{P} = \{\mathsf{eq}(X, X)\}$. Now, "$d_1\ d_1$" is a $(\mathcal{D}, \mathcal{Q}, \mathcal{P})$ chain. To see this, assume that $A = \mathsf{p}(\mathsf{s}(\mathsf{s}(0)), 0)$. Then $\theta_0 = \{X/\mathsf{s}(0), Y/0\}$, $\sigma_0 = \{Z/\mathsf{s}(0)\}$, and $\theta_1 = \{X/0, Y/0\}$.

In this section we show how to synthesize a DT problem from a termination graph built for a logic program with cut such that termination of the DT problem implies termination of the original program w.r.t. the set of queries for which the termination graph was constructed. This approach is far more powerful than first constructing the cut-free logic program as in Sect. 4 and then transforming it into a DT problem. Indeed, the latter approach would fail for our leading example (as the cut-free program (6)-(8) is not terminating), whereas the termination proof succeeds when generating DT problems directly from the termination graph.

Like in the transformation into definite logic programs from [18], we have to prove that there is no derivation of the original program which corresponds to a

[6] For simplicity, we use a set of initial queries instead of a general call set as in [17].

path traversing the cycles in the termination graph infinitely often.

To this end, we build a set $\mathcal{D}$ of DTs for paths in the graph corresponding to cycles and a set $\mathcal{P}$ of program clauses for paths corresponding to the evaluation of intermediate goals. For the component $\mathcal{Q}$ of the resulting DT problem, we use a set of queries based on the root node.

We now illustrate how to use this idea to prove termination of Ex. 1 by building a DT problem for the termination graph from Fig. 3. We again represent each node by a fresh predicate symbol with the different variables occurring in the node as arguments. However, as before, for an INSTANCE node we take the predicate symbol of its child instead where we apply the matching substitution used for the respective instantiation and we do not introduce any predicates for SUC nodes. But in contrast to Sect. 4 and [18], we use different predicates for DTs and program clauses. In this way, we can distinguish between atoms used to represent the traversal of cycles and atoms used as intermediate goals.

To this end, instead of clause paths we now define *triple paths* (that are used to build the component $\mathcal{D}$ of the resulting DT problem) and *program paths* (that are used for the component $\mathcal{P}$ of the DT problem). Triple paths lead from the root node or the successor of an INSTANCE node to the beginning of a cycle, i.e., to an INSTANCE node or the successor of an INSTANCE node where we do not traverse other INSTANCE nodes or their children. Compared to the clause paths of Def. 1, triple paths do not start or stop in left successors of SPLIT nodes, but in contrast they may traverse them. Since finite computations are irrelevant for building infinite chains, triple paths do not stop in SUC nodes either.

Thus, we have two triple paths from the root node A to the INSTANCE nodes F and H. We also have to consider intermediate goals, but this time we use a predicate symbol p_A for the intermediate goal and a different predicate symbol q_A for the DTs. Hence, we obtain $\mathsf{q}_\mathrm{A}(\mathsf{s}(T_6), T_5) \leftarrow \mathsf{q}_\mathrm{A}(T_6, U)$ and $\mathsf{q}_\mathrm{A}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{p}_\mathrm{A}(T_6, T_7), \mathsf{q}_\mathrm{A}(T_7, T_8)$.

Concerning the evaluation for left successors of SPLIT nodes, we build program clauses for the component $\mathcal{P}$ of the DT problem. The clauses result from *program paths* in the termination graph. These are paths starting in a left successor of a SPLIT node and ending in a SUC node. However, in addition to the condition that we do not traverse INSTANCE nodes or their successors, such a path may also not traverse another left successor of a SPLIT node as we are only interested in completely successful evaluations. Thus, the right successor of a SPLIT node must be reached. As the evaluation for left successors of SPLIT nodes may also traverse cycles before it reaches a fact, we also have to consider paths starting in the left successor of a SPLIT node or the successor of an INSTANCE node and ending in an INSTANCE node, a successor of an INSTANCE node, or a SUC node. Compared to the clause paths of Def. 1, the only difference is that program paths do not stop in left successors of SPLIT nodes. Hence, we have two program paths from the root node A to the only SUC node D and to the INSTANCE node H. We also have to consider intermediate goals for the constructed clauses. Thus, we result in the fact $\mathsf{p}_\mathrm{A}(0, 0)$ and the clause $\mathsf{p}_\mathrm{A}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{p}_\mathrm{A}(T_6, T_7), \mathsf{p}_\mathrm{A}(T_7, T_8)$.

So we obtain the DT problem $(\mathcal{D}_G, \mathcal{Q}_G, \mathcal{P}_G)$ for the termination graph G from Fig. 3 where $\mathcal{D}_G$ contains the DTs

$$\mathsf{q}_\mathsf{A}(\mathsf{s}(T_6), T_5) \leftarrow \mathsf{q}_\mathsf{A}(T_6, U).$$
$$\mathsf{q}_\mathsf{A}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{p}_\mathsf{A}(T_6, T_7), \mathsf{q}_\mathsf{A}(T_7, T_8).$$

and $\mathcal{P}_G$ consists of the following clauses.

$$\mathsf{p}_\mathsf{A}(0,0).$$
$$\mathsf{p}_\mathsf{A}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{p}_\mathsf{A}(T_6, T_7), \mathsf{p}_\mathsf{A}(T_7, T_8).$$

Hence, there are three differences compared to the program (6)-(8) we obtain following Def. 2: (i) we do not obtain the fact $\mathsf{q}_\mathsf{A}(0,0)$ for the dependency triples; (ii) we do not obtain the clause $\mathsf{p}_\mathsf{A}(\mathsf{s}(T_6), T_5) \leftarrow \mathsf{p}_\mathsf{A}(T_6, U)$; and (iii) we use p_A instead of q_A in the intermediate goal of the second dependency triple. The latter two differences are essential for success on this example as a ground "input" for p_A on the first argument guarantees a ground "output" on the second argument. Note that this is not the case for the program according to Def. 2.

In our example, $\mathcal{Q}_G$ contains all queries $\mathsf{q}_\mathsf{A}(t_1, t_2)$ where t_1 is ground. Then this DT problem is easily shown to be terminating by our automated termination prover AProVE (or virtually any other tool for termination analysis of definite logic programs by proving termination of $\mathcal{D}_G \cup \mathcal{P}_G$ for the set of queries $\mathcal{Q}_G$).

Now we formally define how to obtain a DT problem from a termination graph. To this end, we first need the notions of triple and program paths to characterize those paths in the termination graph from which we generate the DTs and clauses for the DT problem.

Definition 3 (Triple Path, Program Path). *A path $\pi = n_1 \dots n_k$ in G is a* triple path *iff $k > 1$ and the following conditions are satisfied:*

- $n_1 \in \mathit{Succ}(1, \textsc{Instance}(G))$ *or* n_1 *is the root of* G,
- $n_k \in \textsc{Instance}(G) \cup \mathit{Succ}(1, \textsc{Instance}(G))$,
- *for all* $1 \leq j < k$, *we have* $n_j \notin \textsc{Instance}(G)$, *and*
- *for all* $1 < j < k$, *we have* $n_j \notin \mathit{Succ}(1, \textsc{Instance}(G))$.

A path $\pi = n_1 \dots n_k$ in G is a program path *iff $k > 1$ and the following conditions are satisfied:*

- $n_1 \in \mathit{Succ}(1, \textsc{Instance}(G) \cup \textsc{Split}(G))$,
- $n_k \in \textsc{Suc}(G) \cup \textsc{Instance}(G) \cup \mathit{Succ}(1, \textsc{Instance}(G))$,
- *for all* $1 \leq j < k$, *we have* $n_j \notin \textsc{Instance}(G)$,
- *for all* $1 < j < k$, *we have* $n_j \notin \mathit{Succ}(1, \textsc{Instance}(G))$, *and*
- *for all* $1 < j \leq k$, *we have* $n_j \notin \mathit{Succ}(1, \textsc{Split}(G))$.

Now, we define the DT problem $(\mathcal{D}_G, \mathcal{Q}_G, \mathcal{P}_G)$ for a termination graph G. The set $\mathcal{D}_G$ contains clauses for all triple paths, the queries $\mathcal{Q}_G$ contain all instances represented by the root node, and $\mathcal{P}_G$ contains clauses for all program paths.

Definition 4 (DT Problem from Termination Graph). *Let G be a termination graph whose root is $(f(T_1, \dots, T_m), (\{T_{i_1}, \dots, T_{i_n}\}, \varnothing))$. The DT problem*

$(\mathcal{D}_G, \mathcal{Q}_G, \mathcal{P}_G)$ is defined by $\mathcal{D}_G = \bigcup_{\pi \text{ triple path in } G} Triple(\pi)$, $\mathcal{Q}_G = \{\mathsf{q}_n(t_1, \ldots, t_m) \mid t_{i_1}, \ldots, t_{i_k}$ are ground$\}$ where q_n is the fresh predicate chosen for the root node by Ren_t, and $\mathcal{P}_G = \bigcup_{\pi \text{ program path in } G} Clause(\pi)$.

For a path $\pi = n_1 \ldots n_k$, we define $Clause(\pi) = Ren(n_1)\sigma_\pi \leftarrow I_\pi, Ren(n_k)$ and $Triple(\pi) = Ren_t(n_1)\sigma_\pi \leftarrow I_\pi, Ren_t(n_k)$. Here, Ren and Ren_t are defined as in Def. 2 but Ren_t uses q_n instead of p_n for any node n.

We now state the central theorem of this paper where we prove that termination of the resulting DT problem implies termination of the original logic program with cut for the set of queries represented by the root state of the termination graph. For the proof we refer to the appendix.

Theorem 1 (Correctness). *If G is a proper termination graph for a logic program $\mathcal{P}$ such that $(\mathcal{D}_G, \mathcal{Q}_G, \mathcal{P}_G)$ is terminating, then all concrete states represented by G's root node have only finite derivations w.r.t. the inference rules of Fig. 1.*

6 Implementation and Experiments

We implemented the new transformation in our fully automated termination prover AProVE and tested it on all 402 examples for logic programs from the Termination Problem Data Base (TPDB) [22] used for the annual international Termination Competition [21]. We compared the implementation of the new transformation (AProVE DT) with the implementation of the previous transformation into definite logic programs from [18] (AProVE Cut), and with a direct transformation into term rewrite systems ignoring cuts (AProVE Direct) from [16]. We ran the different versions of AProVE on a 2.67 GHz Intel Core i7 and, as in the international Termination Competition, we used a timeout of 60 seconds for each example. For all versions we give the number of examples which could be proved terminating (denoted "Successes"), the number of examples where termination could not be shown ("Failures"), the number of examples for which the timeout of 60 seconds was reached ("Timeouts"), and the total runtime ("Total") in seconds. For those examples where termination could be proved, we indicate how many of them contain cuts. For the details of this empirical evaluation and to run the three versions of AProVE on arbitrary examples via a web interface, we refer to `http://aprove.informatik.rwth-aachen.de/eval/cutTriples/`.

	AProVE Direct	AProVE Cut	AProVE DT
Successes	243	259	**315**
– with cut	10	78	**82**
– without cut	233	181	**233**
Failures	144	129	**77**
Timeouts	15	14	**10**
Total	2485.7	3288.0	**2311.6**

Table 1. Experimental results on the Termination Problem Data Base

As shown in Table 1, the new transformation significantly increases the number of examples that can be proved terminating. In particular, we obtain 56 additional proofs of termination compared to the technique of [18]. And indeed, for all examples where AProVE Cut succeeds, AProVE DT succeeds, too. Note that while [18] is very successful on examples with cut, its performance is significantly worse than that of AProVE Direct on the other examples of the TPDB.

While we conjecture that our new improved transformation is *always* more powerful than the transformation from [18], a formal proof of this conjecture is not straightforward. The reason is that the clause paths of [18] differ from the triple and program paths in our new transformation. Hence we cannot compare the transformed problems directly.

In addition to being more powerful, the new version using dependency triples is also more efficient than any of the two other versions, resulting in fewer timeouts and a total runtime that is less than the one of the direct version and only 70% of the version corresponding to [18]. However, AProVE DT sometimes spends more time on failing examples, as the new transformation may result in DT problems where the termination proof fails later than for the logic programs resulting from [18].

7 Conclusion

We have shown that the termination graphs introduced by [18] can be used to obtain a transformation from logic programs with cut to dependency triple problems. Our experiments show that this new approach is both considerably more powerful and more efficient than a translation to definite logic programs as in [18]. As the dependency triple framework allows a modular and flexible combination of arbitrary termination techniques from logic programming and even term rewriting, the new transformation to dependency triples can be used as a frontend to any termination tool for logic programs (by taking the union of $\mathcal{D}_G$ and $\mathcal{P}_G$ in the resulting DT problem $(\mathcal{D}_G, \mathcal{Q}_G, \mathcal{P}_G)$) or term rewriting (by using the transformation of [17]).

References

1. K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, London, 1997.
2. T. Arts and J. Giesl. Termination of Term Rewriting using Dependency Pairs. *Theoretical Computer Science*, 236(1,2):133–178, 2000.
3. M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination Analysis of Logic Programs through Combination of Type-Based Norms. *ACM Transactions on Programming Languages and Systems*, 29(2):Article 10, 2007.
4. M. Codish, V. Lagoon, and P. J. Stuckey. Testing for Termination with Monotonicity Constraints. In *ICLP '05*, volume 3668 of *LNCS*, pages 326–340, 2005.
5. D. De Schreye and S. Decorte. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming*, 19,20:199–260, 1994.
6. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer, New York, 1996.

7. J. Giesl. Termination of Nested and Mutually Recursive Algorithms. *Journal of Automated Reasoning*, 19:1–29, 1997.
8. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *LPAR '04*, volume 3452 of *LNAI*, pages 301–331, 2005.
9. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
10. N. Hirokawa and A. Middeldorp. Automating the Dependency Pair Method. *Information and Computation*, 199(1,2):172–199, 2005.
11. J. M. Howe and A. King. Efficient Groundness Analysis in Prolog. *Theory and Practice of Logic Programming*, 3(1):95–124, 2003.
12. M. Marchiori. Proving Existential Termination of Normal Logic Programs. In *AMAST '96*, volume 1101 of *LNCS*, pages 375–390, 1996.
13. F. Mesnard and A. Serebrenik. Recurrence with Affine Level Mappings is P-Time Decidable for CLP(R). *Theory and Practice of Logic Programming*, 8(1):111–119, 2007.
14. M. T. Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. Termination Analysis of Logic Programs based on Dependency Graphs. In *LOPSTR '07*, volume 4915 of *LNCS*, pages 8–22, 2008.
15. M. T. Nguyen, D. De Schreye, J. Giesl, and P. Schneider-Kamp. Polytool: Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. *Theory and Practice of Logic Programming*, 2010. To appear.
16. P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting. *ACM Transactions on Computational Logic*, 10(1), 2009.
17. P. Schneider-Kamp, J. Giesl, and M. T. Nguyen. The Dependency Triple Framework for Termination of Logic Programs. In *LOPSTR '09*, volume 6037 of *LNCS*, pages 37–51, 2010.
18. P. Schneider-Kamp, J. Giesl, T. Ströder, A. Serebrenik, and R. Thiemann. Automated Termination Analysis for Logic Programs with Cut. In *ICLP '10, Theory and Practice of Logic Programming*, 10(4-6):365–381, 2010. Extended version with experimental details appeared as Technical Report AIB-2010-10, RWTH Aachen. Available from `http://aib.informatik.rwth-aachen.de/`.
19. A. Serebrenik and D. De Schreye. On Termination of Meta-Programs. *Theory and Practice of Logic Programming*, 5(3):355–390, 2005.
20. T. Ströder. Towards Termination Analysis of Real Prolog Programs. Diploma Thesis, RWTH Aachen, 2010. `http://aprove.informatik.rwth-aachen.de/eval/cutTriples/`.
21. The Termination Competition. `http://www.termination-portal.org/wiki/Termination_Competition`.
22. The Termination Problem Data Base 7.0 (December 11, 2009). `http://termcomp.uibk.ac.at/status/downloads/`.
23. C. Walther. On Proving the Termination of Algorithms by Machine. *Artificial Intelligence*, 71(1):101–157, 1994.

A Proving the Correctness of the Transformation

For proving the correctness of our new transformation, we use the full definition of DT problems built from a termination graph as opposed to the simplified version we presented in the paper. Here, we must additionally consider the case that some substitutions along a path must be disregarded due to backtracking. We excluded such cases in the paper to ease presentation. To see the problem arising from such a situation, consider the following example.

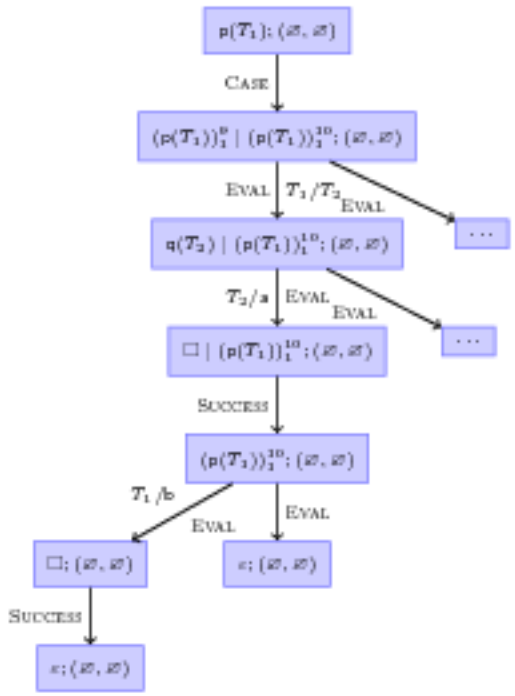
Example 2. Consider the following simple Prolog program $\mathcal{P}$

$$\mathsf{p}(X) \leftarrow \mathsf{q}(X). \tag{9}$$

$$\mathsf{p}(\mathsf{b}). \tag{10}$$

$$\mathsf{q}(\mathsf{a}). \tag{11}$$

and the set of queries $\mathcal{Q} = \{\mathsf{p}(t) \mid \mathsf{p}(t) \in \mathcal{T}(\Sigma, \mathcal{N})\}$. We can obtain the following path as part of a termination graph for $\mathcal{P}$ and $\mathcal{Q}$.



If we successively apply all substitutions along this path, the resulting substitution is $[T_1/\mathsf{a}]$. This is not what we want since the answer substitution for the complete path is $[T_1/\mathsf{b}]$. The problem is that we first reach a Suc node for the answer substitution $[T_1/\mathsf{a}]$ before we reach the second Suc node for the answer substitution $[T_1/\mathsf{b}]$. To detect the latter, we make use of a scope value d. Starting at the Suc node with a scope value of ∞ we update this value along

the EVAL edge to the scope of the EVAL node while gathering the substitution $[T_1/\mathsf{b}]$. Thus, for the next EVAL edge we only gather substitutions for ground terms as the current scope value is not less than the one for the EVAL node. So here, we only gather *id*. By the same reasoning we only gather *id* along the next edge for the first EVAL node. Altogether, we obtain the intended substitution $[T_1/\mathsf{b}]$.

The formal definition of DT problems built for termination graphs is, thus, given as follows.

Definition 5 (DT Problem from Termination Graph). *Let G be a termination graph whose root n is $(f(T_1,\ldots,T_m),(\{T_{i_1},\ldots,T_{i_k}\},\varnothing))$. The DT problem $DTP(G)$ for a termination graph $G=(V,E)$ constructed from a program $\mathcal{P}$ and set of queries $\mathcal{Q}$ is defined as $(\mathcal{D}_G,\mathcal{Q}_G,\mathcal{P}_G)$ where $\mathcal{D}_G=\bigcup_{\pi \text{ triple path in } G} Triple(\pi)$, $\mathcal{Q}_G=\{\mathsf{q}_n(t_1,\ldots,t_m)\mid t_{i_1},\ldots,t_{i_k} \text{ are ground}\}$ with q_n the fresh predicate chosen for the root node by Ren_t, and $\mathcal{P}_G=\bigcup_{\pi \text{ clause path in } G} Clause(\pi)$.*

For a path $\pi=n_1\ldots n_k$, we define $Clause(\pi)=Ren(n_1)\sigma_\pi \leftarrow I_\pi, Ren(n_k)$ and $Triple(\pi)=Ren_t(n_1)\sigma_\pi \leftarrow I_\pi, Ren_t(n_k)$. Here, Ren and Ren_t are defined as in Def. 2 but Ren_t uses q_n for a node n instead of p_n. Furthermore, $\sigma_{\pi,d}$ and I_π are defined as follows (where the substitutions μ, σ, $\sigma|_\mathcal{G}$ and the index m are from the corresponding node n_{j-1}):

$$\sigma_{n_1\ldots n_j,d}=\begin{cases} id & \text{if } j=1\\ \sigma_{n_1\ldots n_{j-1},d}\mu & \text{if } n_{j-1}\in \textsc{Split}(G),\ n_j=Succ(2,n_{j-1})\\ \sigma_{n_1\ldots n_{j-1},m}\sigma & \text{if } n_{j-1}\in \textsc{Eval}(G),\\ & n_j=Succ(1,n_{j-1}), d>m\\ \sigma_{n_1\ldots n_{j-1},d}\sigma|_\mathcal{G} & \text{if } n_{j-1}\in \textsc{Eval}(G)\\ & n_j=Succ(1,n_{j-1}), d\le m\\ \sigma_{n_1\ldots n_{j-1},d} & \text{otherwise}\end{cases}$$

$$I_{n_j\ldots n_k}=\begin{cases}\square & \text{if } j=k\\ Ren(Succ(1,n_j))\sigma_{n_j\ldots n_k,\emptyset}, I_{n_{j+1}\ldots n_k} & \text{if } n_j\in \textsc{Split}(G),\\ & n_{j+1}=Succ(2,n_j)\\ I_{n_{j+1}\ldots n_k} & \text{otherwise}\end{cases}$$

We also state the formal definition of a concretization, which is a substitution of the abstract variables respecting the information from a knowledge base $(\mathcal{G},\mathcal{U})$.

Definition 6 (Concretization). *A substitution γ is a* concretization *w.r.t. $(\mathcal{G},\mathcal{U})$ iff $Dom(\gamma)=\mathcal{A}$, $\mathcal{V}(Range(\gamma))\subseteq\mathcal{N}$, $\mathcal{V}(Range(\gamma|_\mathcal{G}))=\varnothing$, and $s\gamma \not\simeq s'\gamma$ for all $(s,s')\in\mathcal{U}$. The set of concretizations of an abstract state $(S;KB)$ is $\mathcal{CON}(S;KB)=\{S\gamma\mid \gamma \text{ is a concretization w.r.t. } KB\}$.*

The following lemmata show how the DTs and clauses of $DTP(G)$ can be used to simulate derivations of concrete states using the concrete inference rules from Fig. 1 described by the root state of the graph. Before we state the lemmata, we introduce the notions of a state prefix and extension respectively which will be used in the following proofs.

Definition 7 (State Prefix, State Extension). *Let S be a state with $S = S_1 \mid \cdots \mid S_k$ where $\forall i \in \{1, \ldots, k\} : S_i$ is a single state element. Let S' be another state. S is a* state prefix *of S' iff there is a bijection $f : \mathbb{N} \to \mathbb{N}$ and $S' = S'_1 \mid \cdots \mid S'_k \mid S''$ for some state S'' where we have for all $i \in \{1, \ldots, k\}$:*

- *$S_i = ?_m$ implies $S'_i = ?_{f(m)}$*
- *$S_i = \square$ implies $S_i = S'_i$*
- *$S_i = Q$ implies $S'_i = Q', Q''$ for a sequence of terms Q'' where $Q' = Q[!_i/!_{f(i)} \forall i \in \mathbb{N}]$*
- *$S_i = (Q)^n_m$ implies $S'_i = (Q', Q'')^n_{f(m)}$ for a sequence of terms Q'' where $Q' = Q[!_i/!_{f(i)} \forall i \in \mathbb{N}]$*

For two states S and S', S' is a state extension *of S iff S is a state prefix of S'.*

Example 3. Consider the state $S = t_1, t_2 \mid (t_3)^i_m$. The state t_1 is a state prefix of S while the state $t_1, t_2 \mid (t_3)^i_m \mid (t_4)^{i'}_{m'}$ is a state extension of S.

The notions of a state prefix and extension respectively are useful to describe the connection between a termination graph and the concrete derivations it represents. Due to the splitting of backtracking sequences and goals with the rules Parallel and Split, the concrete derivation may contain states which are not represented by only one abstract state, but by several different abstract states instead. Still, we have to take this difference into account when we prove the correctness of our transformation.

Thus, for the simulation of concrete derivations by abstract derivations, we need to follow not only linear paths, but *tree paths* in a termination graph. This is also due to the splitting of goals by the Split rule and to the splitting of backtracking sequences we encounter at Parallel nodes. The following definition therefore gives us a structure for describing the way of a concrete derivation through a termination graph.

Definition 8 (Tree Path). *For termination graph $G = (V, E)$ we call a (possibly infinite) word $\pi = (n_0, v_0, p_0), (n_1, v_1, p_1), (n_2, v_2, p_2), \ldots$ over the set $\mathbb{N} \times V \times (\mathbb{N} \cup \{none\})$ a* tree path *w.r.t. G iff the following conditions are satisfied for all $i, j \in \mathbb{N}$:*

- *$p_0 = none$,*
- *$n_i = n_j \implies i = j$,*
- *$p_i = none \implies i = 0$,*
- *$p_i \in \{n_0, n_1, n_2, \ldots\}$ if $i > 0$,*
- *$n_i = p_j \implies (v_i, v_j) \in E$ and*
- *$p_i < n_i$*

– *there are indices* $i_0, \ldots, i_{m_i} \in \{n_0, n_1, n_2, \ldots\}$ *with* $i_{m_i} = 0$, $i_0 = i$ *and* $p_{i_{r-1}} = n_{i_r}$ *for all* $r \in \{1, \ldots, m_i\}$.

We call (n_i, v_i, p_i) *a leaf of* π *iff there is no* $(n_j, v_j, p_j) \in \pi$ *with* $p_j = n_i$. *For* (n_i, v_i, p_i) *and* (n_j, v_j, p_j) *we call* (n_i, v_i, p_i) *an ancestor of* (n_j, v_j, p_j) *iff there are indices* $i_0, \ldots, i_{m_i} \in \{n_0, n_1, n_2, \ldots\}$ *with* $i_{m_i} = i$, $i_0 = j$ *and* $p_{i_{r-1}} = n_{i_r}$ *for all* $r \in \{1, \ldots, m_i\}$.

To really follow a complete concrete derivation we would have to fork on PARALLEL nodes, but as we will be interested in the relevant parts of the concrete derivations for the reached states only, we may skip the failing branches due to backtracking. Thus, the only nodes where we have to fork our tree path are SPLIT nodes.

Lemma 1 (Success Tree for Concrete Derivations in Termination Graph). *Let* $S\gamma \in \mathcal{CON}(S; KB)$ *with* $S; KB = n \in V$ *for a termination graph* $G = (V, E)$ *and there is a concrete derivation with* l *steps from* $S\gamma$ *to a state* S''. *Then there is a node* $n' \in V$, *a concretization* γ' *and a variable renaming* ρ *on* $\mathcal{N}$ *with* $n' = S'; KB'$, $S'\gamma' \in \mathcal{CON}(S'; KB')$, $S'\gamma'\rho$ *is a state prefix of* S'' *and there is a tree path* $\pi = (0, v_0, p_0), \ldots, (k, v_k, p_k)$ *w.r.t.* G *with the following properties*:

– $v_0 = n$
– *for all* $i \in \{0, \ldots, k\}$ *there are concretizations* γ_i *and variable renamings* ρ_i *on* $\mathcal{N}$ *such that the concrete derivation reaches a state extension of* $S_i\gamma_i\rho_i$ *in* $l_i \leq l$ *steps where* $v_i = S_i; KB_i$ *and* $S_i\gamma_i \in \mathcal{CON}(S_i; KB_i)$
– *for all leaves* (i, v_i, p_i) *of* π *with* $i \neq k$ *we have* $v_i \in \text{SUC}(G)$
– *for all* (i, v_i, p_i) *with more than one successor in* π, *we have* $v_i \in \text{SPLIT}(G)$
– *for all* (i, v_i, p_i) *with* $v_i \in \text{SPLIT}(G)$ *and only one successor* (j, v_j, i) *in* π, *we have* $v_j = Succ(1, v_i)$
– $v_k = n'$

Proof. We perform the proof by induction over the lexicographic combination of first the length l of the concrete derivation and second the edge relation of G'. Here, G' is like G except that it only contains outgoing edges of INSTANCE-, PARALLEL-, and SPLIT-nodes. Note that this induction relation is indeed well-founded as G' is an acyclic and finite graph. The reason is that when traversing nodes $(S; KB)$ in G' the number of terms in S cannot increase. Since this number is strictly decreased in PARALLEL- and SPLIT-nodes any infinite path in G' must in the end only traverse INSTANCE-nodes. This is in contradiction to the definition of termination graph which disallows cycles consisting only of INSTANCE-nodes.

We first show that the lemma holds for nodes $S; KB$ where one of the abstract rules INSTANCE, PARALLEL, or SPLIT have been applied. Here, whenever we have to define the concretization γ' and the variable renaming ρ and if these are not specified then $\gamma' = \gamma$ and $\rho = id$.

– If we applied the INSTANCE rule to n, we have $Succ(1, n) = S'; KB'$ with $S = S'\mu$. From the soundness proof for INSTANCE we know that there is a

concretization γ'' such that $S'\gamma'' \in \mathcal{CON}(S'; KB')$ and $S\gamma = S'\gamma''\mu|_{\mathcal{N}}$. As $\mu|_{\mathcal{N}}$ is a variable renaming we conclude that the concrete derivation from $S\gamma$ to a state extension of S'' can be completely simulated by a corresponding concrete derivation from $S'\gamma''$ to a state extension of S''' of length l where the only difference is the application of $\mu|_{\mathcal{N}}$. To be more precise, if S_i is the i-th state in the concrete derivation from $S\gamma$ to a state extension of S'' then there also is an i-th state S'_i in the concrete derivation from $S'\gamma''$ to a state extension of S''' and $S'_i\mu|_{\mathcal{N}} = S_i$. Hence, we can use the induction hypothesis for the latter concrete derivation to obtain a tree path π' with root $S'; KB'$. To obtain π from π' we first modify all variable renamings by additionally adding $\mu|_{\mathcal{N}}$ ($\rho_i = \rho'_i\mu|_{\mathcal{N}}$). Then we add the node $S; KB$ as new root and start the path with the edge from $S; KB$ to S', KB'.

- If we applied the Parallel rule to n, we reach two states $S_1; KB$ and $S_2; KB$ where $S = S_1 \mid S_2$. There are two cases depending on whether the concrete derivation reaches a state extension of $S_2\gamma$. If the concrete derivation reaches such a state, we use $Succ(2, n)$ instead of n and insert the path from n to $Succ(2, n)$ before the tree path we obtain by the induction hypothesis for $Succ(2, n)$. If the concrete derivation does not reach such a state, we know from the soundness proof of Parallel that a state prefix of S'' must be reachable from $S_1\gamma$ and as we clearly have that S_1 is a state prefix of S, we use $Succ(1, n)$ instead of n and insert the path from n to $Succ(1, n)$ before the tree path we obtain for $Succ(1, n)$ by the induction hypothesis.
- If we applied the Split rule to n, we know that $S = t, Q$, $Succ(1, n) = t; KB$ and $Succ(2, n) = Q\mu; KB'$.
  If the concrete derivation reaches a state extension of $Q\gamma\mu'$ for some answer substitution μ', we know from the soundness proof of Split that there is a concretization γ' w.r.t. KB' such that $Q\gamma\mu' = Q\mu\gamma'$. Additionally, we know that the concrete derivation reaches a state extension of $\square$ from $t\gamma$. As this concrete derivation is shorter than the one of $(t, Q)\gamma$ we obtain a node $n'' \in \textsc{Suc}(G)$ and a tree path π' for $Succ(1, n)$ by the induction hypothesis. Also, we obtain a node n''' and a tree path π'' for $Succ(2, n)$ by the induction hypothesis for the concrete derivation of $(Q\mu)\gamma'$ to a state extension of S''. Using γ' and id for $Succ(2, n)$, we obtain the node $n' = n'''$ and the desired tree path π by using n as the root with π' as its left and π'' as its right subtree path.
  If the concrete derivation does not reach a state extension of $Q\gamma\mu'$ for any answer substitution μ', we know by the soundness proof of Split that a state prefix of S'' must be reachable from $t\gamma$ within l steps. Hence, we can apply the induction hypothesis and add $(S; KB)$ as a new root with only one edge to $(t; KB)$.

For $l = 0$ we know that $S\gamma = S'' \in \mathcal{CON}(S; KB)$. Thus, for $\gamma_0 = \gamma$, $\rho_0 = id$ and $n' = n$ we obtain $\pi = (0, n, none)$ as the desired tree path. So, let $l > 0$. We now perform a case analysis over the first concrete inference rule applied in the concrete derivation where we can assume that none of the abstract rules Instance, Parallel, or Split have been applied to the abstract state.

- For CASE we have $S = t, Q \mid S_r$ and the concrete derivation starts with $S\gamma \vdash (t,Q)^{i_1}_j\gamma \mid \cdots \mid (t,Q)^{i_m}_j\gamma \mid S_r\gamma$. In the abstract setting it remains to analyze an application of the CASE rule to n where we reach the state $n'' = (t,Q)^{i_1}_j \mid \cdots \mid (t,Q)^{i_m}_j \mid S_r; KB$. By the induction hypothesis we obtain a node n''' and a tree path π' with the properties in Lemma 1 for n''. We obtain the desired node $n' = n'''$ and the tree path π by inserting the path from n to n'' before π'.
- For SUC we have $S = \Box \mid S_r$ and the concrete derivation reaches the state $S_r\gamma$. So the only applicable remaining abstract inference rule for n is SUC. Then we reach the state $S_r; KB$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 1 for $S_r; KB$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $S_r; KB$ before π'.
- For FAIL and CUT the proof is analogous to the case where the SUC rule is the first rule in the concrete derivation.
- For EVAL we have $S = (t,Q)^i_j \mid S_r$ and the concrete derivation reaches the state $B'_i\sigma'', Q\gamma\sigma'' \mid S_r\gamma$ as defined in the EVAL rule. We know that the only remaining applicable abstract inference rule for n is EVAL. Then we have $Succ(1,n) = B'_i\sigma, Q\sigma \mid S_r\sigma|_{\mathcal{G}}; KB'$. From the soundness proof of EVAL we know that there is a concretization γ'' w.r.t. KB' with $B'_i\sigma\gamma'', Q\sigma\gamma'' \mid S_r\sigma|_{\mathcal{G}}\gamma'' = B'_i\sigma'', Q\gamma\sigma'' \mid S_r\gamma$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 1 for $B'_i\sigma, Q\sigma \mid S_r\sigma|_{\mathcal{G}}; KB'$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $B'_i\sigma, Q\sigma \mid S_r\sigma|_{\mathcal{G}}; KB'$ before π' using γ'' and id for $B'_i\sigma, Q\sigma \mid S_r\sigma|_{\mathcal{G}}; KB'$.
- For BACKTRACK we have $S = (t,Q)^i_j \mid S_r$ and the concrete derivation reaches the state $S_r\gamma$. Thus, the only remaining applicable abstract inference rules for n are EVAL and BACKTRACK.
  If we applied the EVAL rule we have $Succ(2,n) = S_r; KB'$ as defined in EVAL where we know by the soundness proof of EVAL that γ is a concretization w.r.t. KB'. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 1 for $S_r; KB'$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $S_r; KB'$ before π' using γ and id for $S_r; KB'$.
  If we applied the BACKTRACK rule we have $Succ(1,n) = S_r; KB'$ and, hence, the same case for $Succ(1,n)$ here as for $Succ(2,n)$ in the case of EVAL.

Lemma 2 (Single Concretization). *Given a path $\pi = n_1 \ldots n_k$ with $n_j \notin$ INSTANCE(G) for all $j \in \{1,\ldots,k-1\}$ and a concrete derivation such that there are variable renamings $\rho_1,\ldots,\rho_k$ and concretizations $\gamma_1,\ldots,\gamma_k$ w.r.t. $KB_1,\ldots,KB_k$ where $n_i = S_i; KB_i$ for all $i \in \{1,\ldots,k\}$ and the concrete derivation goes from a state extension of $S_1\gamma_1\rho_1$ to a state extension of $S_k\gamma_k\rho_k$ by reaching state extensions of all $S_i\gamma_i\rho_i$, then there is a variable renaming ρ and a concretization γ w.r.t. all knowledge bases KB_i such that $S_i\gamma_i\rho_i = S_i\gamma\rho$.*

Proof. We perform the proof by induction over the length k of the path π.

For $k = 1$ we have $n_1 = n_k$ and only one variable renaming and concretization $\gamma_1\rho_1 = \gamma\rho$. Hence, the lemma trivially holds.

For $k > 1$ we can assume the lemma holds for paths of length at most $k-1$. By inspection of all abstract inference rules other than INSTANCE we know that only fresh abstract variables are introduced by these rules. We perform a case analysis over n_1 and n_2.

- If $n_1 \in \textsc{Split}(G)$ and $n_2 = \mathit{Succ}(2, n_1)$, i.e., we traverse the right child of a SPLIT node, we have $n_1 = t, Q; KB$ and $n_2 = Q\mu; KB'$ as defined in the SPLIT rule. By the induction hypothesis we obtain a variable renaming ρ and a concretization γ' w.r.t. KB_j for all $j \in \{2, \ldots, k\}$ such that $S_j\gamma_j\rho_j = S_j\gamma'\rho$. In particular, we have $Q\mu\gamma_2\rho_2 = Q\mu\gamma'\rho$. By [18, Lemma 5] and the fact that the concrete derivation reaches a state extension of $Q\mu\gamma_2\rho_2$ from a state extension of $(t, Q)\gamma_1\rho_1$ with some answer substitution μ', we obtain $\gamma_1\rho_1\mu' = \mu\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(KB)}$ and $\rho_1 = \rho_2$. Since only fresh abstract variables are introduced along π, we have for all abstract variables $T \in (\mathcal{A}(t) \cup \mathcal{A}(Q) \cup \mathcal{A}(KB)) \setminus (\mathcal{A}(Q\mu) \cup \mathcal{A}(KB'))$ that $T \notin \mathcal{A}(S_j) \cup \mathcal{A}(KB_j)$. Hence, we can define the concretization γ by $T\gamma = T\gamma_1$ for $T \in (\mathcal{A}(t) \cup \mathcal{A}(Q) \cup \mathcal{A}(KB)) \setminus (\mathcal{A}(Q\mu) \cup \mathcal{A}(KB'))$ and $T\gamma = T\gamma'$ otherwise. Then we obviously have $S_i\gamma\rho = S_i\gamma_i\rho_i$ for all $i \in \{1, \ldots, k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2, \ldots, k\}$. As γ is equally defined to γ' for all variables occurring in the knowledge bases KB_j, we clearly have that γ is a concretization w.r.t. KB_j. Moreover, as γ is equally defined to γ_1 for all variables occurring in KB_1, it is also a concretization w.r.t. KB_1.
- If $n_1 \in \textsc{Eval}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse the left child of an EVAL node, we have $n_1 = \langle t, Q\rangle^c_m \mid S; KB$ and $n_2 = B'_c\sigma, Q\sigma \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the EVAL rule. By the induction hypothesis we obtain a variable renaming ρ and a concretization γ' w.r.t. KB_j for all $j \in \{2, \ldots, k\}$ such that $S_j\gamma_j\rho_j = S_j\gamma'\rho$. In particular, we have $B'_c\sigma\gamma_2\rho_2, Q\sigma\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2 = B'_c\sigma\gamma'\rho, Q\sigma\gamma'\rho \mid S\sigma|_{\mathcal{G}}\gamma'\rho$. By [18, Lemma 2] and the fact that the concrete derivation reaches a state extension of $B'_c\sigma\gamma_2\rho_2, Q\sigma\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2$ from a state extension of $\langle t, Q\rangle^c_m\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with answer substitution σ', we obtain $\gamma_1\rho_1\sigma' = \sigma\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$ and $\rho_1 = \rho_2$. Since only fresh abstract variables are introduced along π, we have for all abstract variables $T \in (\mathcal{A}(t) \cup \mathcal{A}(Q) \cup \mathcal{A}(S) \cup \mathcal{A}(KB)) \setminus (\mathcal{A}(B'_c\sigma) \cup \mathcal{A}(Q\sigma) \cup \mathcal{A}(S\sigma|_{\mathcal{G}}) \cup \mathcal{A}(KB'))$ that $T \notin \mathcal{A}(S_j) \cup \mathcal{A}(KB_j)$. Hence, we can define the concretization γ by $T\gamma = T\gamma_1$ for $T \in (\mathcal{A}(t) \cup \mathcal{A}(Q) \cup \mathcal{A}(S) \cup \mathcal{A}(KB)) \setminus (\mathcal{A}(B'_c\sigma) \cup \mathcal{A}(Q\sigma) \cup \mathcal{A}(S\sigma|_{\mathcal{G}}) \cup \mathcal{A}(KB'))$ and $T\gamma = T\gamma'$ otherwise. Then we obviously have $S_i\gamma\rho = S_i\gamma_i\rho_i$ for all $i \in \{1, \ldots, k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2, \ldots, k\}$. As γ is equally defined to γ' for all variables occurring in the knowledge bases KB_j, we clearly have that γ is a concretization w.r.t. KB_j. Moreover, as γ is equally defined to γ_1 for all variables occurring in KB_1, it is also a concretization w.r.t. KB_1.
- For all other cases we know that $\gamma_1\rho_1 = \gamma_2\rho_2$. Hence, the lemma follows by the induction hypothesis.

Lemma 3 (Answer Substitutions are Instances of Path Substitutions). *Given a path $\pi = n_1 \dots n_k$ with $n_j \notin \textsc{Instance}(G)$ for all $j \in \{1, \dots, k-1\}$ and a concrete derivation such that there is a variable renaming ρ and a concretization γ w.r.t. $KB_1, \dots, KB_k$ where $n_i = S_i; KB_i$ for all $i \in \{1, \dots, k\}$ and the concrete derivation goes from a state extension of $S_1\gamma\rho$ to a state extension of $S_k\gamma\rho$ with answer substitution δ by reaching state extensions of all $S_i\gamma\rho$, then $\sigma_{\pi,\infty}\gamma\rho = \gamma\rho\delta$ and $S_k\gamma\rho\delta = S_k\gamma\rho$.*

Proof. We perform the proof by induction over the length k of the path π.

For $k = 1$ we have $n_1 = n_k$ and the empty answer substitution $\delta = id = \sigma_{n_1,\infty}$. Hence, the lemma trivially holds.

For $k > 1$ we can assume the lemma holds for paths of length at most $k-1$. We perform a case analysis over n_1 and n_2.

- If $n_1 \in \textsc{Split}(G)$ and $n_2 = Succ(2, n_1)$, i.e., we traverse the right child of a Split node, we have $n_1 = t, Q; KB$ and $n_2 = Q\mu; KB'$ as defined in the Split rule. By the induction hypothesis we obtain $\sigma_{n_2 \dots n_k,\infty}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete derivation from a state extension of $Q\mu\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. For the answer substitution μ' of the concrete derivation from a state extension of $(t, Q)\gamma\rho$ to a state extension of $Q\mu\gamma\rho$ we know by [18, Lemma 5] that $\gamma\rho\mu' = \mu\gamma\rho$. Therefore, we have $\gamma\rho\delta = \gamma\rho\mu'\delta'' = \mu\gamma\rho\delta'' = \mu\sigma_{n_2 \dots n_k,\infty}\gamma\rho = \sigma_{\pi,\infty}\gamma\rho$. Furthermore, we know that μ is idempotent as all variables in the range of μ are fresh. As we applied μ to S_2 already and we know by inspection of the abstract inference rules other than Instance that only fresh variables are introduced along π, we obtain $S_k\mu = S_k$. Hence, we have $S_k\gamma\rho\delta = S_k\gamma\rho\mu'\delta'' = S_k\mu\gamma\rho\delta'' = S_k\gamma\rho\delta'' = S_k\gamma\rho$.
- If $n_1 \in \textsc{Eval}(G)$ and $n_2 = Succ(1, n_1)$, i.e., we traverse the left child of an Eval node, we have $n_1 = (t, Q)^c_m \mid S; KB$ and $n_2 = B'_c\sigma, Q\sigma \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the Eval rule. By the induction hypothesis we obtain $\sigma_{n_2 \dots n_k,\infty}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete derivation from a state extension of $B'_c\sigma\gamma\rho, Q\sigma\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. For the answer substitution σ' of the concrete derivation from a state extension of $(t, Q)^c_m\gamma\rho \mid S\gamma\rho$ to a state extension of $B'_c\sigma\gamma\rho, Q\sigma\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ we know by [18, Lemma 2] that $\gamma\rho\sigma' = \sigma\gamma\rho$ and $\gamma\rho = \sigma|_{\mathcal{G}}\gamma\rho$. Furthermore, we know that σ is idempotent as the range of σ contains only fresh variables. Now there are two cases depending on whether $\sigma_{\pi,\infty}$ starts with σ or $\sigma|_{\mathcal{G}}$. In the first case we know by definition of $\sigma_{\pi,\infty}$ that we do not have a node $n_j \in \textsc{Eval}(G)$ with $j \in \{2, \dots, k-1\}$ with a scope less or equal than m. Since the scopes are ascendingly ordered, we know that the concrete derivation did not backtrack the substitution σ'. Hence, we obtain $\gamma\rho\delta = \gamma\rho\sigma'\delta'' = \sigma\gamma\rho\delta'' = \sigma\sigma_{n_2 \dots n_k,\infty}\gamma\rho = \sigma_{\pi,\infty}\gamma\rho$. Additionally, we already applied σ to S_2. As we know by inspection of all abstract inference rules other than Instance that only fresh variables are introduced along π, we obtain $S_k\sigma = S_k$ by σ being idempotent. Hence, we have $S_k\gamma\rho\delta = S_k\gamma\rho\sigma'\delta'' = S_k\sigma\gamma\rho\delta'' = S_k\gamma\rho\delta'' = S_k\gamma\rho$. In the second case we know by definition of $\sigma_{\pi,\infty}$ that we do have a node $n_j \in \textsc{Eval}(G)$

with $j \in \{2, \ldots, k-1\}$ with a scope less or equal than n. Since the scopes are ascendingly ordered, we know that the concrete derivation did backtrack the substitution σ' and we have the same answer substitution δ'' for the complete concrete derivation. This amounts to $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{n_2 \ldots n_k,\infty}\gamma\rho = \sigma_{\pi,\infty}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.
- For all other cases we know that the concrete derivation has the empty answer substitution from the state extension of $S_1\gamma\rho$ to the state extension of $S_2\gamma\rho$. By the induction hypothesis we obtain $\sigma_{n_2 \ldots n_k,\infty}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete derivation from a state extension of $S_2\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. Then we have $\gamma\rho\delta = \gamma\rho\delta' = \sigma_{n_2 \ldots n_k,\infty}\gamma\rho = \sigma_{\pi,\infty}\gamma\rho$ and $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

Now, using the preceding results, we can simulate the evaluation of intermediate goals with the clauses from $\mathcal{P}_G$. We use the connection between concrete and abstract derivations by tree paths and the connection between the answer substitutions of successful evaluations and the substitutions we generated from a path in a termination graph.

Lemma 4 (Simulation of Intermediate Goals Using $\mathcal{P}_G$). *Let $S; KB \in Succ(1, \textsc{Instance}(G) \cup \textsc{Split}(G))$.*

If there is a variable renaming ρ on $\mathcal{N}$ and a concretization γ w.r.t. KB such that there is a concrete derivation from $S\gamma\rho$ to a state extension of $\square$, then we have $Ren(S; KB)\gamma\rho \vdash^*_{\mathcal{P}_G} \square$.

Proof. From Lemma 1 we know that there is a tree path $\pi_{tree} = (0, v_0, none), (1, v_1, p_1), \ldots, (k, v_k, p_k)$ with the following properties:

- $v_0 = S; KB$
- for all $i \in \{0, \ldots, k\}$ there are concretizations γ_i and variable renamings ρ_i on $\mathcal{N}$ such that the concrete derivation reaches a scope variant of a state extension of $S_i\gamma_i\rho_i$ in $l_i \leq l$ steps where $v_i = S_i; KB_i$ and $S_i\gamma_i \in \mathcal{CON}(S_i; KB_i)$
- for all (i, v_i, p_i) with more than one successor in π_{tree}, we have $v_i \in \textsc{Split}(G)$
- for all (i, v_i, p_i) with $v_i \in \textsc{Split}(G)$ and only one successor (j, v_j, i) in π_{tree}, we have $v_j = Succ(1, v_i)$
- for all leaves (i, v_i, p_i) of π_{tree} we have $v_i \in \textsc{Suc}(G)$

The last property follows from the fact that the concrete derivation reaches a state extension of $\square$.

We perform the proof by induction over the length k of the tree path π_{tree} which is at least 1.

For $k = 1$ we have $\pi_{tree} = (0, S; KB, none)$ and $S; KB \in \textsc{Suc}(G)$. Thus, we have $Ren(S; KB) = \square \vdash^*_{\mathcal{P}_G} \square$. For $k > 1$ we can assume the lemma holds for tree paths of length at most $k-1$.

Consider the rightmost path π through the tree path π_{tree}. As the concrete derivation reaches a state extension of $\square$, π must be a sequence of clause paths w.r.t. G. If this sequence has the length 1, we obtain $Ren(S; KB)\sigma_{\pi,\infty} \leftarrow I_\pi \in \mathcal{P}_G$ and by Lemma 2 we can w.l.o.g. assume that all concretizations and variable renamings used in π are equal to γ and ρ. By Lemma 3 we know that

$Ren(S;KB)\sigma_{\pi,\infty}$ unifies with $Ren(S;KB)\gamma\rho$ by $\gamma\rho\delta$ where δ is the answer substitution of the concrete derivation from $S\gamma\rho$ to a state extension of $\square$. Thus, we obtain $Ren(S;KB)\gamma\rho \vdash^*_{\mathcal{P}_{cr}} I_\pi\gamma\rho\delta$. As we have for all intermediate goals in I_π some subtree paths in π such that the concrete derivation reaches a state extension of $\square$ from the respective intermediate goal in $I_\pi\gamma\rho$, we can use the induction hypothesis and obtain $I_\pi\gamma\rho \vdash^*_{\mathcal{P}_{cr}} \square$. Now, since δ is the answer substitution for the complete concrete derivation, we obtain $I_\pi\gamma\rho\delta \vdash^*_{\mathcal{P}_{cr}} \square$ by Lemma 3. Now let the length of the sequence be greater than 1. For the first clause path in the sequence to a node v_i we obtain $Ren(S;KB)\gamma\rho \vdash^*_{\mathcal{P}_{cr}} Ren(v_i)\gamma\rho$ by the identical argument as for the sequence of length 1. The remaining tree path from (i, v_i, p_i) is shorter than k and, therefore, we obtain $Ren(v_i)\gamma\rho \vdash^*_{\mathcal{P}_{cr}} \square$ by the induction hypothesis.

As we can simulate intermediate goals with $\mathcal{P}_G$, we can simulate the evaluation along triple paths with the DTs in $DTP(G)$.

Lemma 5 (Simulation of Concrete Derivations Using $DTP(G)$). *Given a concrete derivation and a corresponding tree path π_{tree} w.r.t. G with the properties from Lemma 1 and a rightmost path $\pi = n_0 \ldots n_k$ in π_{tree} where for $i < k$, $n_i \notin$ INSTANCE(G), we have $Ren_t(n_0)\gamma_0\rho_0 \vdash_{Triple(\pi)} \circ \vdash^*_{\mathcal{P}_{cr}} Ren_t(n_k)\gamma_k\rho_k$ where the γ_j and ρ_j are the concretizations and variable renamings respectively used in π_{tree}.*

Proof. We perform the proof by induction over k, which is the length of π minus one.

For $k = 0$ we have $n_0 = n_k$ and, therefore, $Triple(\pi) = Ren_t(n_0) \leftarrow Ren_t(n_0)$. Thus, obviously we have $Ren_t(n_0)\gamma_0\rho_0 \vdash_{Triple(\pi)} Ren_t(n_0)\gamma_0\rho_0$. For $k > 0$, we can assume the lemma holds for paths of length at most k. By Lemma 2 we can w.l.o.g. assume that $\gamma_0 = \ldots = \gamma_k = \gamma$ and $\rho_0 = \ldots = \rho_k = \rho$.

We now perform a case analysis based on n_k and n_{k-1}.

If $n_{k-1} \in$ SPLIT(G) and $n_k = Succ(2, n_{k-1})$, i.e., we traverse the right child of a SPLIT node, we know by the induction hypothesis that $Ren_t(n_0)\gamma\rho \vdash_{Triple(n_1 \ldots n_{k-1}) \circ \vdash^*_{\mathcal{P}_{cr}}} Ren_t(n_{k-1})\gamma\rho$. From the [18, Lemma 5] we know that $\gamma\rho\mu' = \mu\gamma\rho$ where $Ren(Succ(1, n_{k-1}))\gamma\rho \vdash^*_{\mathcal{P}_{cr}} \square$ with answer substitution μ'. The latter follows from Lemma 4 and the fact that the concrete derivation must reach a state extension of $\square$ from $Succ(1, n_{k-1})\gamma\rho$. By Lemma 3, we also know that $Ren_t(n_0)\gamma\rho$ unifies with the head of $Triple(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete derivation and that we do not have to apply δ to $Ren_t(n_k)$ completely. Furthermore, we obtain the same derivation for $Ren(Succ(1, n_{k-1}))\gamma\rho\delta$ as for $Ren(Succ(1, n_{k-1}))\gamma\rho$, but with the empty answer

substitution by Lemma 3. Thus, we obtain:

$$\begin{array}{rl} & Ren_t(n_0)\gamma\rho \\ \vdash_{Triple(\pi)} \circ \vdash^*_{\mathcal{P}_{cr}} & Ren(Succ(1,n_{k-1}))\gamma\rho\delta, Ren_t(n_k)\mu^{-1}\gamma\rho\mu' \\ \vdash^*_{\mathcal{P}_{cr}} & Ren_t(n_k)\mu^{-1}\gamma\rho\mu' \\ = & Ren_t(n_k)\mu^{-1}\mu\gamma\rho \\ = & Ren_t(n_k)\gamma\rho \end{array}$$

If $n_{k-1} \in \textsc{Eval}(G)$ and $n_k = Succ(1, n_{k-1})$, i.e., we traverse the left child of an EVAL node, we know by the induction hypothesis that $Ren_t(n_0)\gamma\rho \vdash_{Triple(n_1 \ldots n_{k-1})} \circ \vdash^*_{\mathcal{P}_{cr}} Ren_t(n_{k-1})\gamma\rho$. From [18, Lemma 2] we know that $Q\gamma\rho\sigma' = Q\sigma\gamma\rho$ and $S\gamma\rho = S\sigma|_{\mathcal{G}}\gamma\rho$. By Lemma 3, we also know that $Ren_t(n_0)\gamma\rho$ unifies with the head of $Triple(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete derivation and that we do not have to apply δ to $Ren_t(n_k)$ completely. Let $n_{k-1} = (t, Q)^i_m \mid S; (\mathcal{G}, \mathcal{U})$. Then, we obtain:

$$\begin{array}{rl} & Ren_t(n_0)\gamma\rho \\ \vdash_{Triple(\pi)} \circ \vdash^*_{\mathcal{P}_{cr}} & Ren_t(Q\gamma\rho\sigma' \mid S\gamma\rho; (\mathcal{G}, \mathcal{U})) \\ = & Ren_t(Q\sigma\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho; (\mathcal{G}', \mathcal{U}\sigma|_{\mathcal{G}})) \\ = & Ren_t(n_k)\gamma\rho \end{array}$$

If $n_{k-1} \in \textsc{Eval}(G)$ and $n_k = Succ(2, n_{k-1})$ or $n_{k-1} \in \textsc{Backtrack}(G) \cup \textsc{Suc}(G) \cup \textsc{Fail}(G)$, i.e., we backtrack by removing the first element of the backtracking sequence, we know by Lemma 3 that $Ren_t(n_0)\gamma\rho$ unifies with the head of $Triple(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete derivation and that we do not have to apply δ to $Ren_t(n_k)$ completely. We perform a case analysis based on the existence of a CASE node introducing m in $n_1 \ldots n_{k-1}$. If such a CASE node does not exist in our path, there is also no SPLIT node in our path. Thus, $I_\pi = \square$. By Lemma 3, we also know that $Ren_t(n_0)\gamma\rho$ unifies with the head of $Triple(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete derivation and that we do not have to apply δ to $Ren_t(n_k)$. Thus, we obtain $Ren_t(n_0)\gamma\rho \vdash_{Triple(\pi)} Ren_t(n_k)\gamma\rho$. If there is such a CASE node in our path, i.e., there is a j such that $n_j \in \textsc{Case}(G)$ and $Succ(1, n_j) = (t', Q')^{i'}_m \mid S'; KB_j$, we know by the induction hypothesis that $Ren_t(n_0)\gamma\rho \vdash_{Triple(n_1 \ldots n_j)} \circ \vdash^*_{\mathcal{P}_{cr}} Ren_t(n_j)\gamma\rho$. For the remaining path π' we have $I_{\pi'} = \square$ and $\gamma\rho$ being an instance of $\sigma_{\pi',\infty}$ by Lemma 3. Thus, we obtain $Ren_t(n_0)\gamma\rho \vdash_{Triple(\pi)} \circ \vdash^*_{\mathcal{P}_{cr}} Ren_t(n_k)\gamma\rho$.

Finally, if n_{k-1} is in none of the above sets, we have $\sigma_{\pi,\infty} = \sigma_{n_1 \ldots n_{k-1},\infty}$ and $I_\pi = I_{n_1 \ldots n_{k-1}}$. Again, from the induction hypothesis we know that $Ren_t(n_0)\gamma\rho \vdash_{Triple(n_1 \ldots n_{k-1})} \circ \vdash^*_{\mathcal{P}_{cr}} Ren_t(n_{k-1})\gamma\rho$. From the definition of the abstract rules used, we know that $\mathcal{V}(s_k) \subseteq \mathcal{V}(s_{k-1})$ as they do not apply any substitutions and, thus, $Ren_t(n_0)\gamma\rho \vdash_{Triple(\pi)} \circ \vdash^*_{\mathcal{P}_{cr}} Ren_t(n_k)\gamma\rho$.

Combining these results yields our central theorem.

Proof. Assume $(\mathcal{D}_G, \mathcal{Q}_G, \mathcal{P}_G)$ is terminating, but there is a concretization $S\gamma \in \mathcal{CON}(S; KB)$ from $root(G) = n_0 = S; KB$ that has an infinite concrete derivation. Then, according to Lemma 1 there is an infinite tree path π_{tree} where the rightmost path $\pi = n_0, n_1, n_2, \ldots$ in π_{tree} is an infinite sequence of triple paths $\pi_0, \pi_1, \pi_2, \ldots$ and there are indices $l_0, l_1, l_2, \ldots$ such that $\pi_m = n_{l_m}, \ldots$.

Now, according to Lemma 5 we know that for all $m \in \mathbb{N}$ there are concretizations γ_m and variable renamings ρ_m such that $Ren_t(n_{l_m})\gamma_m\rho_m \vdash_{Triple(\pi_m)} \circ \vdash^*_{\mathcal{P}_G} Ren_t(n_{l_{m+1}})\gamma_{m+1}\rho_{m+1}$.

Thus, $Ren_t(n_{l_0})\gamma_{l_0}\rho_0 \in \mathcal{Q}_G$ starts an infinite $(\mathcal{D}_G, \mathcal{Q}_G, \mathcal{P}_G)$-chain
$(Ren_t(n_{l_0})\sigma_{\pi_0,\mathbb{Z}} \leftarrow I_{\pi_0}, Ren_t(n_{l_1}))$,
$(Ren_t(n_{l_1})\sigma_{\pi_1,\mathbb{Z}} \leftarrow I_{\pi_1}, Ren_t(n_{l_2}))$,
...
which contradicts our initial assumption and, hence, proves the theorem.

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from `http://aib.informatik.rwth-aachen.de/`. To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

2005-01 * Fachgruppe Informatik: Jahresbericht 2004

2005-02 Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"

2005-03 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions

2005-04 Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem

2005-05 Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06 Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information

2005-07 Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks

2005-08 Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut

2005-09 Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures

2005-10 Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11 Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-12 Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems

2005-13 Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14 Felix C. Freiling, Sukumar Ghosh: Code Stabilization

2005-15 Uwe Naumann: The Complexity of Derivative Computation

2005-16 Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

2005-17 Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18 Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"

2005-19 Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers

2005-20 Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.

2005-21 Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited

2005-22 Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins

2005-23 Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves

2005-24 Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks

2006-01 * Fachgruppe Informatik: Jahresbericht 2005

2006-02 Michael Weber: Parallel Algorithms for Verification of Large Systems

2006-03 Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

2006-04 Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation

2006-05 Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F

2006-06 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color

2006-07 Thomas Colcombet, Christof Löding: Transforming structures by set interpretations

2006-08 Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs

2006-09 Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking

2006-10 Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed

2006-11 Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers

2006-12 Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning

2006-13 Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities

2006-14 Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"

2006-15 Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices

2006-16 Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness

2006-17 Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines

2007-01 * Fachgruppe Informatik: Jahresbericht 2006

2007-02 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations

2007-03 Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase

2007-04 Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation

2007-05 Uwe Naumann: On Optimal DAG Reversal

2007-06 Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking

2007-07 Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications

2007-08 Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches

2007-09 Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption

2007-10 Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes

2007-11 Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke

2007-12 Uwe Naumann: An L-Attributed Grammar for Adjoint Code

2007-13 Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs

2007-14 Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes

2007-15 Volker Stolz: Temporal assertions for sequential and concurrent programs

2007-16 Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks

2007-17 René Thiemann: The DP Framework for Proving Termination of Term Rewriting

2007-18 Uwe Naumann: Call Tree Reversal is NP-Complete

2007-19 Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control

2007-20 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems

2007-21 Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains

2007-22 Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets

2008-01 * Fachgruppe Informatik: Jahresbericht 2007

2008-02 Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing

2008-03 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, Harald Zankl: Maximal Termination

2008-04 Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler

2008-05 Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations

2008-06 Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs

2008-07 Alexander Nyßen, Horst Lichter: The MeDUSA Reference Manual, Second Edition

2008-08 George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs

2008-09 George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs

2008-10 George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time

2008-11 George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows

2008-12 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages

2008-13 Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs

2008-14 Bastian Schlich: Model Checking of Software for Microcontrollers

2008-15 Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves

2008-16 Hendrik vorn Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study

2008-17 Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving

2008-18 Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems

2008-19 Dirk Wilking: Empirical Studies for the Application of Agile Methods to Embedded Systems

2009-02 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications

2009-03 Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices

2009-04 Daniel Klünder: Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation

2009-05 George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs

2009-06 George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete

2009-07 Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I

2009-08 Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs

2009-09 Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem

2009-10 Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm

2009-11 Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs

2009-12 Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes

2009-13 Martin Zimmermann: Time-optimal Winning Strategies for Poset Games

2009-14 Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)

2009-15 Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems

2009-16 George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs

2009-17 Carsten Kern: Learning Communicating and Nondeterministic Automata

2009-18 Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies

2010-02 Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time

2010-03 Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering

2010-04 René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme

2010-07 George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms

2010-08 Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting

2010-09 George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs

2010-10 Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut

2010-11 Martin Zimmermann: Parametric LTL Games

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.