RWTH Aachen

Department of Computer Science
Technical Report

Automated Termination Analysis for Logic Programs with Cut

Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2010-10

RWTH Aachen · Department of Computer Science · April 2010

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/

Automated Termination Analysis for Logic Programs with Cut∗

PETER SCHNEIDER-KAMP
Dept. of Mathematics and Computer Science, University of Southern Denmark, Denmark

JÜRGEN GIESL, THOMAS STRÖDER
LuFG Informatik 2, RWTH Aachen University, Germany

ALEXANDER SEREBRENIK
Dept. of Mathematics and Computer Science, TU Eindhoven, The Netherlands

RENÉ THIEMANN
Institute of Computer Science, University of Innsbruck, Austria

Abstract

Termination is an important and well-studied property for logic programs. However, almost all approaches for automated termination analysis focus on definite logic programs, whereas real-world Prolog programs typically use the *cut* operator. We introduce a novel pre-processing method which automatically transforms Prolog programs into logic programs without cuts, where termination of the cut-free program implies termination of the original program. Hence after this pre-processing, any technique for proving termination of definite logic programs can be applied. We implemented this pre-processing in our termination prover AProVE and evaluated it successfully with extensive experiments.

KEYWORDS: automated termination analysis, cut, definite logic programs

1 Introduction

Automated termination analysis for logic programs has been widely studied, see, e.g., (Bruynooghe et al. 2007; Codish et al. 2005; De Schreye and Decorte 1994; Mesnard and Serebrenik 2007; Nguyen et al. 2010; Schneider-Kamp et al. 2009; Serebrenik and De Schreye 2005). Still, virtually all existing techniques only prove universal[1] termination of *definite* logic programs, which do not use the *cut* "!". An exception is (Marchiori 1996), which transforms "safely typed" logic programs to term rewrite systems (TRSs). However, the resulting TRSs are complex and since

∗ Supported by the Deutsche Forschungsgemeinschaft (DFG) under grant GI 274/5-2, the DFG Research Training Group 1298 (*AlgoSyn*), and the Danish Natural Science Research Council.

[1] *Universal* termination means that *all* answers are found after a finite number of derivation steps. Unfortunately, up to now there was hardly any automated technique to prove the at least equally interesting property of *existential* termination (i.e., finite failure or the *first* answer is found after a finite number of derivation steps). Note that if one can handle the cut then one also immediately obtains a method to prove existential termination. The reason is that a query Q is existentially terminating iff $Q, !$ is universally terminating.

there is no implementation of (Marchiori 1996), it is unclear whether they can be handled by existing TRS termination tools. Moreover, (Marchiori 1996)'s method does not allow arbitrary cuts (e.g., it does not operate on programs like Ex. 1).

In the present paper, we introduce a novel approach which shows that universal termination of logic programs with cuts can indeed be proved *automatically* for (typically infinite) classes of queries. This solves an important open problem in automated termination analysis of logic programs.

Example 1

We want to prove termination of the following program for the class of queries $\{\mathsf{div}(t_1, t_2, t_3) \,|\, t_1, t_2 \text{ are ground}\}$. Since we only regard programs without pre-defined predicates, the program contains clauses defining predicates for failure and equality. So the atom $\mathsf{failure(a)}$ always fails and corresponds to Prolog's pre-defined "fail".

$$\mathsf{div}(X,\mathsf{0},Z) \leftarrow !, \mathsf{failure(a)}. \quad (1)$$
$$\mathsf{div}(\mathsf{0},Y,Z) \leftarrow !, \mathsf{eq}(Z,\mathsf{0}). \quad (2)$$
$$\mathsf{div}(X,Y,\mathsf{s}(Z)) \leftarrow \mathsf{sub}(X,Y,U), \mathsf{div}(U,Y,Z). \quad (3)$$
$$\mathsf{failure(b)}. \quad (4)$$
$$\mathsf{eq}(X,X). \quad (5)$$
$$\mathsf{sub}(\mathsf{0},Y,\mathsf{0}). \quad (6)$$
$$\mathsf{sub}(X,\mathsf{0},X). \quad (7)$$
$$\mathsf{sub}(\mathsf{s}(X),\mathsf{s}(Y),Z) \leftarrow \mathsf{sub}(X,Y,Z). \quad (8)$$

Any termination analyzer that ignores the cut fails, as $\mathsf{div}(\mathsf{0},\mathsf{0},Z)$ would lead to the subtraction of $\mathsf{0}$ and start an infinite derivation using Clause (3). So due to the cut, (universal) termination effectively depends on the order of the clauses.

There are already several static analysis techniques for logic programming with cut, e.g., (Filé and Rossi 1993; Mogensen 1996), which are based on abstract interpretation (Cousot and Cousot 1992; Le Charlier et al. 1994; Spoto and Levi 1998). However, these works do not capture termination as an observable and none of these results targets termination analysis explicitly. While we also rely on the idea of abstraction, our approach does not operate directly on the abstraction. Instead, we synthesize a cut-free logic program from the abstraction, such that termination of the derived program implies termination of the original one. Thus, we can benefit from the large body of existing work on termination analysis for cut-free programs. Our approach is inspired by our previous successful technique for termination analysis of Haskell programs (Giesl et al. 2006), which in turn was inspired by related approaches to program optimization (Sørensen and Glück 1995).

In Sect. 2, we introduce the required notions and present a set of simple inference rules that characterize logic programming with cut for concrete queries. In Sect. 3 we extend these inference rules to handle *classes* of queries. Using these rules we can automatically build so-called termination graphs, cf. Sect. 4. Then, Sect. 5 shows how to generate a new cut-free logic program from such a graph automatically.

Of course, one can transform any Turing-complete formalism like logic programming with cuts into another Turing-complete formalism like cut-free logic programming. But the challenge is to develop a transformation such that termination of the resulting programs is *easy to analyze by existing termination tools*. Our implementation and extensive experiments in Sect. 6 show that with our approach, the resulting cut-free program is usually easy to handle by existing tools.

2 Concrete Derivations

See e.g. (Apt 1997) for the basics of logic programming. We distinguish between individual cuts to make their scope explicit. So a signature Σ contains all predicate and function symbols and all labeled versions of the cut $\{!_m/0 \mid m \in \mathbb{N}\}$. 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. To ease the presentation, in the paper we exclude terms with cuts $!_m$ as proper subterms. 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 sequence of terms from $\mathcal{T}(\Sigma, \mathcal{V})$. Let $\mathit{Goal}(\Sigma, \mathcal{V})$ be the set of all such sequences, where $\Box$ is the empty goal.

A *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 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 $s \not\sim t$. 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 $t[f/g]$.

Now we recapitulate the operational semantics of logic programming with cut. Compared to other formulations like (Andrews 2003; Billaud 1990; de Vink 1989; Kulas and Beierle 2000; Spoto 2000), the advantage of our formalization is that it is particularly suitable for an extension to *classes* of queries in Sect. 3 and 4, and for synthesizing cut-free programs in Sect. 5. A formal proof on the correspondence of our inference rules to the semantics of the Prolog ISO standard (Deransart et al. 1996) can be found in (Ströder 2010).

Our semantics is given by 7 inference rules. They operate on *states* which represent the current goal, and also the backtrack information that is needed to describe the effect of cuts. The backtrack information is given by a sequence of goals which are optionally labeled by the program clause that has to be applied to the goal next. Moreover, our states also contain explicit *marks* for the scope of a cut.

Definition 1 (*Concrete State*)
A *concrete state* is a sequence of elements from $\mathit{Goal}(\Sigma, \mathcal{V}) \cup (\mathit{Goal}(\Sigma, \mathcal{V}) \times \mathbb{N} \times \mathbb{N}) \cup \{?_n \mid n \in \mathbb{N}\}$, where elements are separated by “|”. $\mathit{State}(\Sigma, \mathcal{V})$ is the set of all concrete states.

So an element of a state can be $Q \in \mathit{Goal}(\Sigma, \mathcal{V})$; or a labeled goal $Q^i_m \in \mathit{Goal}(\Sigma, \mathcal{V}) \times \mathbb{N} \times \mathbb{N}$ representing that we must apply the i-th program clause to Q next, where m determines how a cut introduced by the body of the i-th clause will be labeled; or $?_m$. Here, $?_m$ serves as a marker to denote the end of the scope of cuts $!_m$ labeled with m. Whenever a cut $!_m$ is reached, all elements preceding $?_m$ are discarded.

Now we express derivations in logic programming with cut by seven rules. Here, S and S' are concrete states and the goal Q may also be $\Box$ (then “t, Q” is t).

Definition 2 (*Semantics with Concrete Inference Rules*)

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

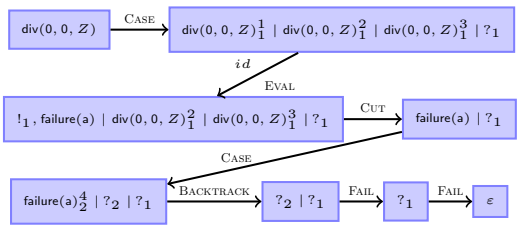
$$\frac{t, Q \mid S}{(t,Q)^{i_1}_m \mid \ldots \mid (t,Q)^{i_k}_m \mid ?_m \mid S}\ (\textsc{Case})$$

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

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

The Suc rule stands for "success" and it is applicable if the first goal of our sequence could be proved. As we handle universal termination, we then have to backtrack to the next goal in the sequence. Fail means that for the current m-th case analysis, there are no further backtracking possibilities. This rule is applicable if $?_m$ is the first element of our backtracking sequence. But the whole derivation does not have to fail, since the state S may still contain further alternative goals which have to be examined.

To make the backtracking possibilities explicit, the resolution of a program clause with the first atom t of the current goal is split into two operations. The Case analysis determines which clauses could be applied to t by slicing the program according to t's root symbol. It replaces the current goal (t, Q) by a goal labeled with the index i_1 of the first such clause and adds copies of (t, Q) labeled by the indices $i_2, \ldots, i_k$ of the other potentially applicable clauses as backtracking possibilities. Note that here, the top-down clause selection rule is taken into account. Additionally, these goals are labeled by a fresh mark $m \in \mathbb{N}$ that is greater than all previous marks, and $?_m$ is added at the end of the new backtracking goals to denote the scope of cuts. For instance, consider the program of Ex. 1 and the query $\mathsf{div}(0, 0, Z)$. This query is represented by the concrete state consisting of just the goal $\mathsf{div}(0, 0, Z)$. Here, we obtain the sequence depicted at the side. The Case rule results in a state which represents a case analysis where we first try to apply the first div-clause (1). When backtracking later on, we use clauses (2) and (3).



For a goal $(t, Q)^i_m$, if t unifies with the head H_i of the corresponding clause, we apply Eval. This rule replaces t by the body B_i of the clause and applies the mgu σ to the result. When depicting rule applications as trees, the corresponding edge is labeled with $\sigma|_{\mathcal{V}(t)}$. All cuts occurring in B_i are labeled with m. The reason is that if one reaches such a cut, then all further alternative goals up to $?_m$ are discarded.

If t does not unify with H_i, we apply the Backtrack rule. Then, Clause i cannot be used and we just backtrack to the next possibility in our backtracking sequence.

Finally, there are two CUT rules. The first rule removes all backtracking information on the level m where the cut was introduced. Since the explicit scope is represented by $!_m$ and $?_m$, we have turned the cut into a *local* operation depending solely on the current state. Note that $?_m$ must not be deleted as the current goal Q could still lead to another cut $!_m$. The second CUT rule is used if $?_m$ is missing (e.g., if a cut $!_m$ is already in the initial query). Later on, such states can also result from the additional PARALLEL inference rule which will be introduced in Sect. 4. We treat such states as if $?_m$ were added at the end of the backtracking sequence.

So to apply the cut $!_1$ in our example, we remove all subsequent elements in the list up to $?_1$. So we remove the backtracking goals $\mathsf{div}(\mathsf{0},\mathsf{0},Z)_1^2$ and $\mathsf{div}(\mathsf{0},\mathsf{0},Z)_1^3$ and obtain the state $\mathsf{failure}(\mathsf{a})_2^4 \mid ?_1$, which eventually fails. Note that due to the cut, we did not have to backtrack using the other div-clauses.

Note that these rules do not overlap, i.e., there is at most one rule that can be applied to any state. The only case where no rule is applicable is when the state is the empty sequence (denoted ε) or when the first goal starts with a variable.

The rules of Def. 2 define the semantics of logic programs with cut using states. They can also be used to define the semantics using derivations between goals: there is a derivation from the goal Q to Q' in the program $\mathcal{P}$ (denoted $Q \vdash^*_{\mathcal{P},\theta} Q'$) iff repeated application of our rules can transform the state[2] Q to a state of the form $\overline{Q}' \mid S$ for some S, and Q' results from $\overline{Q}'$ by removing all labels. Moreover, $\theta = \theta_1\theta_2\ldots\theta_n$ where $\theta_1,\ldots,\theta_n$ are the mgu's used in those applications of the EVAL rule that led to $\overline{Q}'$. We call $\theta_{|\mathcal{V}(Q)}$ the corresponding *answer substitution*. If θ is not of interest, we write $\vdash_{\mathcal{P}}$ instead of $\vdash_{\mathcal{P},\theta}$.

Consequently, our inference rules can be used for termination proofs: If there is an infinite derivation (w.r.t. $\vdash_{\mathcal{P}}$) starting in some goal Q, then there is also an infinite sequence of inference rule applications starting in the state Q, i.e., Q is a "non-terminating state". Note that we distinguish derivations in logic programming (i.e., $Q \vdash_{\mathcal{P}} Q'$ for goals Q and Q') from sequences of states that result from application of the inference rules in Def. 2. If a state S can be transformed into a state S' by such an inference rule, we speak of a "*state-derivation*".

3 Abstract Derivations

To represent *classes* of queries, we introduce *abstract terms*, i.e., terms containing two kinds of variables. Let $\mathcal{A}$ be the set 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 $\mathcal{U} \subseteq \mathcal{T}(\Sigma,\mathcal{V}) \times$

[2] If Q contains cuts, then the inference rules have to be applied to $Q[!/!_1]$ instead of Q.

$\mathcal{T}(\Sigma, \mathcal{V})$. The variables in $\mathcal{G}$ may only be instantiated by ground terms. And $(s, s') \in \mathcal{U}$ means that we are restricted to instantiations γ of the abstract variables where $s\gamma \not\sim s'\gamma$, i.e., s and s' may not become unifiable when instantiating them with γ.

Definition 3 (*Abstract State*)
The set of *abstract states* $AState(\Sigma, \mathcal{N}, \mathcal{A})$ is a set of pairs $(S; KB)$ of a concrete state $S \in State(\Sigma, \mathcal{N} \cup \mathcal{A})$ and a *knowledge base* KB.

A substitution γ is a *concretization* of an abstract state if it respects the knowledge base $(\mathcal{G},\mathcal{U})$. So first, γ instantiates all abstract variables, i.e., $Dom(\gamma) = \mathcal{A}$. Second, when applying γ, the resulting term must be concrete, i.e., $\mathcal{V}(Range(\gamma)) \subseteq \mathcal{N}$. Third, abstract variables from $\mathcal{G}$ may only be replaced by ground terms, i.e., $\mathcal{V}(Range(\gamma_{|\mathcal{G}})) = \varnothing$. Fourth, for all pairs $(s, s') \in \mathcal{U}$, $s\gamma$ and $s'\gamma$ must not unify.

Definition 4 (*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 $s\gamma \not\sim s'\gamma$ for all $(s, s') \in \mathcal{U}$. The set of concretizations of an abstract state $(S; KB)$ is $\mathcal{C}on(S; KB) = \{S\gamma \mid \gamma \text{ is a concretization w.r.t. } KB\}$.

Example 2
Consider the abstract state which consists of the single goal $\mathsf{sub}(T_1, T_2, T_3)$ and the knowledge base $(\{T_1, T_2\}, \{(T_1, T_3)\})$, with $T_i \in \mathcal{A}$ for all i. So here $\mathcal{G} = \{T_1, T_2\}$ and $\mathcal{U}$ only contains (T_1, T_3). This represents all concrete states $\mathsf{sub}(t_1, t_2, t_3)$ where t_1, t_2 are ground terms and where t_1 and t_3 do not unify, i.e., t_3 does not match t_1. For example, $\mathsf{sub}(\mathsf{0}, \mathsf{0}, Z)$ is not represented as $\mathsf{0}$ and Z unify. In contrast, $\mathsf{sub}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), \mathsf{0})$ and $\mathsf{sub}(\mathsf{0}, \mathsf{0}, \mathsf{s}(\mathsf{0}))$ are represented. Note that $\mathsf{sub}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), \mathsf{0})$ can be reduced to $\mathsf{sub}(\mathsf{0}, \mathsf{0}, \mathsf{0})$ using Clause (8) from Ex. 1. But Clause (8) cannot be applied to all concretizations. For example, the concrete state $\mathsf{sub}(\mathsf{0}, \mathsf{0}, \mathsf{s}(\mathsf{0}))$ is also represented by our abstract state, but here no clause is applicable.

Ex. 2 demonstrates that we need to adapt our inference rules to reflect that sometimes a clause can be applied only for some concretizations of the abstract variables, and to exploit the information from the knowledge base of the abstract state. We now adapt our inference rules to abstract states that represent *sets* of concrete states. The invariant of our rules is that all states represented by the parent node are terminating if all the states represented by its children are terminating. (We now also permit rules leading to more than one child node.)

Definition 5 (*Sound Rules*)
An abstract state is called *terminating* iff all its concretizations are terminating. A rule $\rho : AState(\Sigma, \mathcal{N}, \mathcal{A}) \to 2^{AState(\Sigma, \mathcal{N}, \mathcal{A})}$ is *sound* if $(S; KB)$ is terminating whenever all $(S'; KB') \in \rho(S; KB)$ are terminating.

To prove the soundness of our adapted inference rules, we will ensure that they all satisfy the following *simulation property* for every abstract state $(S; KB)$ and each of its concretizations $S\gamma \in \mathcal{C}on(S; KB)$: If the concrete state $S\gamma$ can be transformed into a concrete state R by a concrete inference rule, then the abstract state $(S; KB)$ can be transformed into an abstract state $(S'; KB')$, such that R is a concretization

of the state $(S'; KB')$, i.e., $R \in \mathcal{C}on(S'; KB')$. Note that the simulation property implies the soundness of the abstract rules. If $(S; KB)$ were non-terminating, then one of its concretizations $S\gamma \in \mathcal{C}on(S; KB)$ would also be non-terminating. But by the soundness of the concrete rules, then R (i.e., one of the concretizations of $(S'; KB')$) would be non-terminating as well, which implies non-termination of $(S'; KB')$.

The rules Suc, Fail, Cut, and Case do not change the knowledge base and are, thus, straightforward to adapt. Here, $S \mid S'; KB$ stands for $((S \mid S'); KB)$.

*Definition 6 (Abstract Inference Rules – Part 1 (*Suc, Fail, Cut, Case*))*

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

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

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

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

For Suc, Fail, Cut and Case, again at most one of the rules is applicable. For the concrete Eval and Backtrack rules, in Def. 2, we determined which of these two rules to choose by trying to unify the first atom t with the head H_i of the corresponding clause. But as demonstrated by Ex. 2, in the abstract case we might need to apply Eval for some concretizations and Backtrack for others. Backtrack can be used for *all* concretizations of our abstract state if t does not unify with H_i or if their mgu contradicts $\mathcal{U}$. This gives rise to the abstract Backtrack rule in the following definition. When the abstract Backtrack rule is not applicable, we still cannot be sure that $t\gamma$ unifies with H_i for all concretizations γ. Thus, we have an abstract Eval rule with two successor states that combines both the concrete Eval and the concrete Backtrack rule.

*Definition 7 (Abstract Inference Rules – Part 2 (*Backtrack, Eval*))*

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

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., $\mathcal{V}(\sigma(X))$ only contains fresh abstract variables for all $X \in \mathcal{V}$. Moreover, $\mathcal{G}' = \mathcal{A}(\mathit{Range}(\sigma_{|\mathcal{G}}))$ and $B'_i = B_i[!/!_m]$.

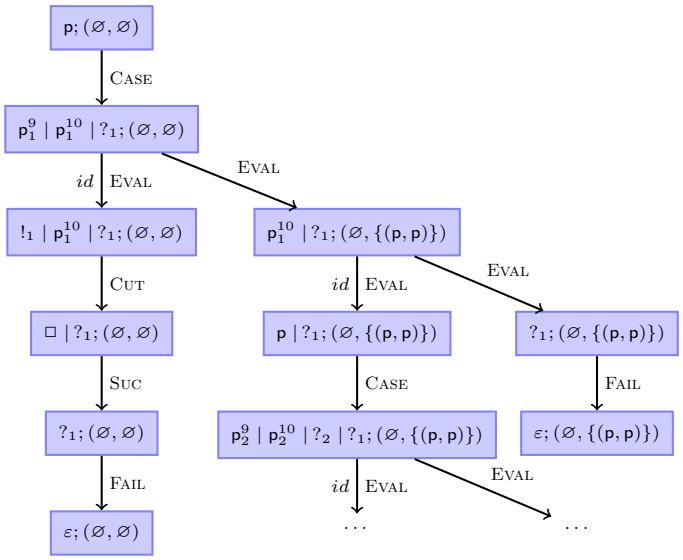
Before explaining the Eval rule in detail, we illustrate the above definition with a first example without variables. The example shows that although the abstract Eval rule has a second child which corresponds to a backtrack step, the additional abstract Backtrack rule is also needed. Otherwise, even if the clause $H_i \leftarrow B_i$ cannot be applied to any concretization of the current abstract goal (e.g., because

of a cut), we would always have to generate *two* children with the EVAL rule (one for the successful application of the clause and one for backtracking). In this way, instead of a short finite tree that ends with failure, we would obtain an infinite tree.

Example 3
To see this, consider the following program:

$$\mathsf{p} \leftarrow \mathsf{!}. \quad (9) \qquad\qquad \mathsf{p} \leftarrow \mathsf{p}. \quad (10)$$

Without the abstract BACKTRACK rule, we would obtain the following tree from the initial state $(\mathsf{p}; (\varnothing, \varnothing))$. Here, the edge to the left child of the EVAL node is marked by the corresponding mgu (which is the identity id in our example).



Note that in the rightmost node marked with "...", one would again apply the EVAL rule etc. which would lead to an infinite tree. The reason is that in the EVAL rule one does not take into account that no concretization of the current abstract goal would unify with any head of a program clause (since EVAL disregards the information in the set $\mathcal{U}$ which in our case states that p may not unify with p; i.e., in our case there is no possible concretization of this abstract state). So instead of the EVAL rule, we should rather apply the BACKTRACK rule to the state $(\mathsf{p}_1^{10} \mid ?_1; (\varnothing, \{(\mathsf{p}, \mathsf{p})\}))$ which would result in the only child $(?_1; (\varnothing, \{(\mathsf{p}, \mathsf{p})\}))$. Another step with the FAIL rule would then give the final state $(\varepsilon; (\varnothing, \{(\mathsf{p}, \mathsf{p})\}))$, i.e., the resulting tree would be finite.

In the abstract EVAL rule, the knowledge base is updated differently for the successors corresponding to the concrete EVAL and to the concrete BACKTRACK rule. For all concretizations corresponding to the second successor of EVAL, the concretization of t does not unify with H_i. Hence, here we add the pair (t, H_i) to the set $\mathcal{U}$.

Now consider concretizations γ where $t\gamma$ and H_i unify, i.e., concretizations γ

corresponding to the first successor of the EVAL rule. Then for any $T \in \mathcal{G}$, $T\gamma$ is a ground instance of $T\sigma$. Hence, we replace all $T \in \mathcal{G}$ by $T\sigma$, i.e., we apply $\sigma_{|\mathcal{G}}$ to $\mathcal{U}$ and S. Now the new set $\mathcal{G}'$ of abstract variables that may only be instantiated by ground terms is $\mathcal{A}(Range(\sigma_{|\mathcal{G}}))$. As before, t is replaced by the instantiated clause body B_i where we label cuts with the number m of the current CASE analysis.

In the EVAL rule, w.l.o.g. we assume that $mgu(t, H_i)$ renames (a) all abstract variables and (b) also all non-abstract variables to fresh abstract variables. As illustrated by the following two examples, this is needed to handle "sharing" effects correctly, i.e., to handle concretizations which introduce multiple occurrences of (concrete) variables.

Example 4
We first illustrate the need for (b), i.e., for the renaming of non-abstract variables into fresh abstract variables. Consider the following program.

$$\mathsf{p}(\mathsf{a}). \ (11) \qquad \mathsf{p}(\mathsf{b}). \ (12) \qquad \mathsf{q}(\mathsf{c}) \leftarrow !. \ (13) \qquad \mathsf{q}(X) \leftarrow \mathsf{q}(X). \ (14)$$

Regard the abstract state $((\mathsf{p}(T), \mathsf{q}(X))_1^{11}; (\varnothing, \varnothing))$ with $T \in \mathcal{A}$ and $X \in \mathcal{N}$. It has the concretization $(\mathsf{p}(X), \mathsf{q}(X))_1^{11}$ which results from instantiating the abstract variable T with the concrete variable X. So now the instantiation of T "shares" the variable X which was already present in the abstract state. Using the concrete EVAL rule, the state $(\mathsf{p}(X), \mathsf{q}(X))_1^{11}$ can be transformed into $\mathsf{q}(\mathsf{a})$. However, if we did not rename concrete variables, then the abstract EVAL rule would only transform the abstract state $((\mathsf{p}(T), \mathsf{q}(X))_1^{11}; (\varnothing, \varnothing))$ into $(\mathsf{q}(X); (\varnothing, \varnothing))$. Now the simulation property would be violated since $\mathsf{q}(\mathsf{a})$ is not a concretization of $(\mathsf{q}(X); (\varnothing, \varnothing))$, as concretizations may not instantiate non-abstract variables like X. In fact, the state $\mathsf{q}(\mathsf{a})$ is non-terminating, whereas $(\mathsf{q}(X); (\varnothing, \varnothing))$ is terminating and thus, the EVAL rule would no longer be sound. In contrast, when using unifiers that rename all concrete variables to fresh abstract variables as in Def. 7, the abstract EVAL rule yields the state $(\mathsf{q}(T'); (\varnothing, \varnothing))$ where T' is a fresh abstract variable.

Example 5
Now we illustrate the need for (a), i.e., for the renaming of abstract variables. We use the same program as in Ex. 4 and consider the abstract state $((\mathsf{p}(X), \mathsf{q}(T))_1^{11} \mid (\mathsf{p}(X), \mathsf{q}(T))_1^{12}; (\varnothing, \varnothing))$. It has the concretization $((\mathsf{p}(X), \mathsf{q}(X))_1^{11} \mid (\mathsf{p}(X), \mathsf{q}(X))_1^{12})$ which results from instantiating the abstract variable T with the concrete variable X. So now also the two instantiations of T "share" the variable X. Since these two instantiations belong to two different backtracking alternatives, it should be possible to instantiate the variable X differently in these two alternatives. Using the concrete EVAL rule, the state $((\mathsf{p}(X), \mathsf{q}(X))_1^{11} \mid (\mathsf{p}(X), \mathsf{q}(X))_1^{12})$ can be transformed into $(\mathsf{q}(\mathsf{a}) \mid (\mathsf{p}(X), \mathsf{q}(X))_1^{12})$. However, if we did not rename abstract variables, then the abstract EVAL rule would transform the abstract state $((\mathsf{p}(X), \mathsf{q}(T))_1^{11} \mid (\mathsf{p}(X), \mathsf{q}(T))_1^{12}; (\varnothing, \varnothing))$ into $(\mathsf{q}(T) \mid (\mathsf{p}(X), \mathsf{q}(T))_1^{12}; (\varnothing, \varnothing))$. Now the simulation property would be violated since $(\mathsf{q}(\mathsf{a}) \mid (\mathsf{p}(X), \mathsf{q}(X))_1^{12})$ is not a concretization of $(\mathsf{q}(T) \mid (\mathsf{p}(X), \mathsf{q}(T))_1^{12}; (\varnothing, \varnothing))$. To solve this problem, we use unifiers that rename all abstract variables into fresh ones. Then the abstract EVAL rule would create the new abstract state $(\mathsf{q}(T') \mid (\mathsf{p}(X), \mathsf{q}(T))_1^{12}; (\varnothing, \varnothing))$ instead.

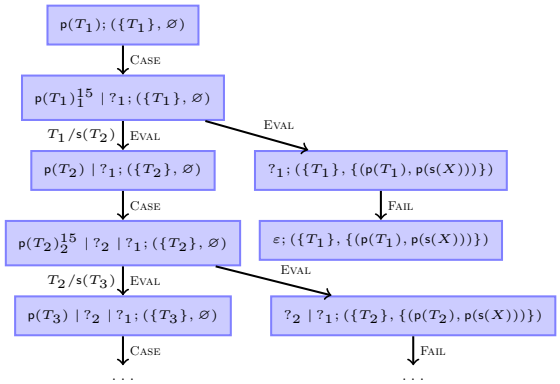
With the rules of Def. 6 and 7, the simulation property is fulfilled. It guarantees that any concrete state-derivation with the rules from Def. 2 can also be simulated with the abstract rules from Definitions 6 and 7.

Now any concrete derivation with the rules from Def. 2 can also be simulated with the abstract rules from Def. 6 and 7. But unfortunately, even for terminating goals, in general these rules yield an infinite tree. The reason is that there is no bound on the size of terms represented by the abstract variables and hence, the abstract EVAL rule can be applied infinitely often.

Example 6
Consider the 1-rule program

$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X). \tag{15}$$

For queries of the form $\mathsf{p}(t)$ where t is ground, the program terminates. However, the tree built using the abstract inference rules is obviously infinite.



4 From Trees to Graphs

To obtain a finite graph instead of an infinite tree, we now introduce an additional INSTANCE rule which allows us to connect the current state $(S;KB)$ with a previous state $(S';KB')$, provided that the current state is an instance of the previous state. In other words, every concretization of $(S;KB)$ must be a concretization of $(S';KB')$. Still, INSTANCE is often not enough to obtain a finite graph.

Example 7
We extend Ex. 6 by the following additional fact.

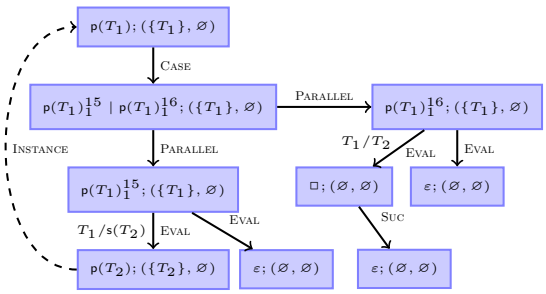
$$\mathsf{p}(X). \tag{16}$$

For queries $\mathsf{p}(t)$ where t is ground, the program still terminates. If we start with $(\mathsf{p}(T_1);(\{T_1\},\varnothing))$, then the CASE rule results in the state $(\mathsf{p}(T_1)_1^{15} \mid \mathsf{p}(T_1)_1^{16} \mid ?_1;(\{T_1\},\varnothing))$ and the EVAL rule produces two new states, one of them being $(\mathsf{p}(T_2) \mid \mathsf{p}(\mathsf{s}(T_2))_1^{16} \mid ?_1;(\{T_2\},\varnothing))$.

To simplify states, from now on we will eliminate so-called *non-active* marks $?_m$ which occur as first or as last element in states. Eliminating $?_m$ from the beginning of a state is possible, as FAIL would also remove such a $?_m$. Eliminating $?_m$ from the end of a state is possible, as applying the first CUT rule to a state ending in $?_m$ is equivalent to applying the second CUT rule to the same state without $?_m$.

We will also reduce the knowledge base to just those abstract variables that occur in the state and remove pairs (s,s') from $\mathcal{U}$ where $s \not\sim s'$. Still, $(\mathsf{p}(T_2) \mid \mathsf{p}(\mathsf{s}(T_2))_1^{16};(\{T_2\},\varnothing))$ is not an instance of the previous state $(\mathsf{p}(T_1);(\{T_1\},\varnothing))$ due to the added backtrack goal $\mathsf{p}(\mathsf{s}(T_2))_1^{16}$. In other words, as soon as there is more than one clause for some predicate p, then each application of the CASE rule produces

an additional backtracking target. For this reason, the new state is not an instance of any previous state. Therefore, we now introduce a PARALLEL rule that allows us to split a backtracking sequence into separate problems. Now we obtain the graph on the right.



Clearly, PARALLEL may transform terminating into non-terminating states. For example, in the program with the clause $\mathsf{p} \leftarrow \mathsf{p}$, the state $(!_1 \mid \mathsf{p};(\varnothing,\varnothing))$ is terminating, but the PARALLEL rule could transform it into $(!_1;(\varnothing,\varnothing))$ and the non-terminating state $(\mathsf{p};(\varnothing,\varnothing))$. But without further conditions, PARALLEL is not only "incomplete", but also unsound. Consider a state $(!_2 \mid !_1 \mid ?_2 \mid \mathsf{p};(\varnothing,\varnothing))$ for the program $\mathsf{p} \leftarrow \mathsf{p}$. The state is not terminating, as $!_1$ is not reachable. Thus, one eventually evaluates p. But if one splits the state into $(!_2;(\varnothing,\varnothing))$ and $(!_1 \mid ?_2 \mid \mathsf{p};(\varnothing,\varnothing))$, both new states terminate. So the problem is that due to the splitting of the backtracking sequence, we can suddenly reach the cut $!_1$ that is unreachable in reality and thereby we can cut away the non-terminating part p.

To solve this problem, in addition to the "active marks" (cf. Ex. 7) we introduce the notion of *active cuts*. The active cuts of a state S are those $m \in \mathbb{N}$ where $!_m$ occurs in S or where $!_m$ can be introduced by EVAL applied to a labeled goal $(t,q)_m^i$ occurring in S. Now the PARALLEL rule may only split a backtracking sequence into two parts S and S' if the active cuts of S and the active marks of S' are disjoint.

Definition 8 (*Abstract Inference Rules – Part 3 (*INSTANCE, PARALLEL*)*)

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

$$\frac{S \mid S';KB}{S;KB \qquad S';KB}\ (\textsc{Parallel}) \qquad \text{if } AC(S) \cap AM(S') = \varnothing$$

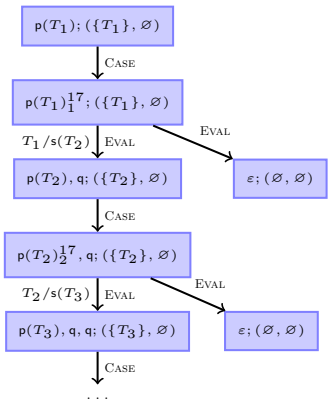
The *active cuts* $AC(S)$ are all m where $!_m$ is in S or $(t,q)_m^i$ is in S and c_i's body has a cut. The *active marks* $AM(S)$ are all m where $S = S' \mid ?_m \mid S''$ and $S' \neq \varepsilon \neq S''$.

Example 8

However, there are still examples where the graph cannot be "closed". Consider the program

$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X), \mathsf{q}. \quad (17) \qquad \mathsf{q}. \quad (18)$$

For queries $\mathsf{p}(t)$ where t is ground, the program again terminates. With Def. 6, 7, and 8, we obtain the infinite tree on the right. It never encounters



an instance of a previous state, since each resolution with Clause (17) adds a q to the goal.

Thus, we introduce a final abstract SPLIT rule to split a state $(t, Q; KB)$ into $(t; KB)$ and a state $(Q\mu; KB')$, where μ approximates the answer substitutions for t. The edge from $(t, Q; KB)$ to $(Q\mu; KB')$ is labeled with $\mu|_{\mathcal{V}(t)\cup\mathcal{V}(Q)}$. To simplify the SPLIT rule, we only define it for backtracking sequences of one element. To obtain such a sequence, we can use the PARALLEL rule.

*Definition 9 (Abstract Inference Rules – Part 4 (*SPLIT*))*

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

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

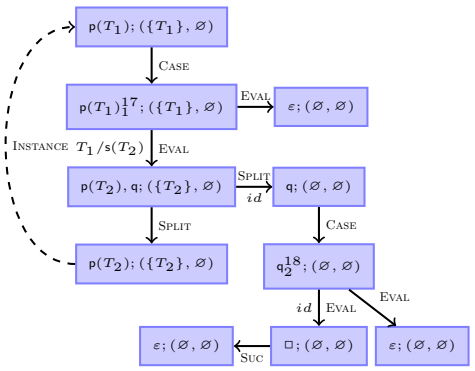
Here, *ApproxGnd* is defined as follows. We assume that we have a *groundness analysis* function $\mathit{Ground}_\mathcal{P} : \Sigma \times 2^\mathbb{N} \to 2^\mathbb{N}$, see, e.g., (Howe and King 2003). If p is an n-ary predicate, $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$, and $\mathit{Ground}_\mathcal{P}(p, \{i_1, \ldots, i_m\}) = \{j_1, \ldots, j_k\}$, then any successful derivation $p(t_1, \ldots, t_n) \vdash^*_{\mathcal{P},\theta} \square$ where $t_{i_1}, \ldots, t_{i_m}$ are ground will lead to an answer substitution θ such that $t_{j_1}\theta, \ldots, t_{j_k}\theta$ are ground. So $\mathit{Ground}_\mathcal{P}$ approximates which positions of p will become ground if the "input" positions $i_1, \ldots, i_m$ are ground. Now if $t = p(t_1, \ldots, t_n)$ is an abstract term where $t_{i_1}, \ldots, t_{i_m}$ are ground in every concretization (i.e., all their variables are from $\mathcal{G}$), then $\mathit{ApproxGnd}(t, \mu)$ returns the μ-renamings of all abstract variables that will be ground in every successful derivation starting from a concretization of t. Thus, $\mathit{ApproxGnd}(t, \mu)$ contains the abstract variables of $t_{j_1}\mu, \ldots, t_{j_k}\mu$. So formally

$$\mathit{ApproxGnd}(p(t_1, \ldots, t_n), \mu) = \{\mathcal{A}(t_j\mu) \mid j \in \mathit{Ground}_\mathcal{P}(p, \{i \mid \mathcal{V}(t_i) \subseteq \mathcal{G}\})\}$$

Example 9

To illustrate Def. 9, regard the program of Ex. 1 and the state $(\mathsf{sub}(T_5, T_6, T_8), \mathsf{div}(T_8, T_6, T_7); (\{T_5, T_6\}, \mathcal{U}))$ with $T_5, T_6, T_7, T_8 \in \mathcal{A}$. (This state will occur in the termination proof of div, cf. Ex. 10.) We have $\mathcal{G} = \{T_5, T_6\}$ and hence if $\mathsf{sub}(t_1, t_2, t_3)$ is $\mathsf{sub}(T_5, T_6, T_8)$, then $\mathit{Ground}_\mathcal{P}(\mathsf{sub}, \{i \mid \mathcal{V}(t_i) \subseteq \mathcal{G}\}) = \mathit{Ground}_\mathcal{P}(\mathsf{sub}, \{1, 2\}) = \{1, 2, 3\}$. In other words, if the first two arguments of sub are ground and the derivation is successful, then the answer substitution also instantiates the third argument to a ground term. Since μ only renames variables outside of $\mathcal{G}$, we have $\mu = \{T_7/T_9, T_8/T_{10}\}$.

So $\mathit{ApproxGnd}(\mathsf{sub}(T_5, T_6, T_8), \mu) = \{\mathcal{A}(t_1\mu), \mathcal{A}(t_2\mu), \mathcal{A}(t_3\mu)\} = \{T_5\mu, T_6\mu, T_8\mu\} = \{T_5, T_6, T_{10}\}$. So the SPLIT rule transforms the current state to $(\mathsf{sub}(T_5, T_6, T_8); (\{T_5, T_6\}, \mathcal{U}))$ and $(\mathsf{div}(T_{10}, T_6, T_9); (\{T_5, T_6, T_{10}\}, \mathcal{U}\mu))$ where one can eliminate T_5 from the new groundness set $\mathcal{G}'$.



With the additional SPLIT rule, we can always obtain finite graphs instead of infinite trees. (This will be

proved in Thm. 2.) Thus, no further rules are needed. As depicted on the previous page, now we can also close the graph for Ex. 8's program.

Thm. 1 proves the soundness of all our abstract inference rules. In other words, if all children of a node are terminating, then the node is terminating as well.

Theorem 1 (*Soundness of the Abstract Inference Rules*)
The inference rules from Def. 6, 7, 8, and 9 are sound.[3]

5 From Termination Graphs to Logic Programs

Now we introduce *termination graphs* as a subclass of the graphs obtained by Def. 6, 7, 8, 9. Then we show how to extract cut-free programs from termination graphs.

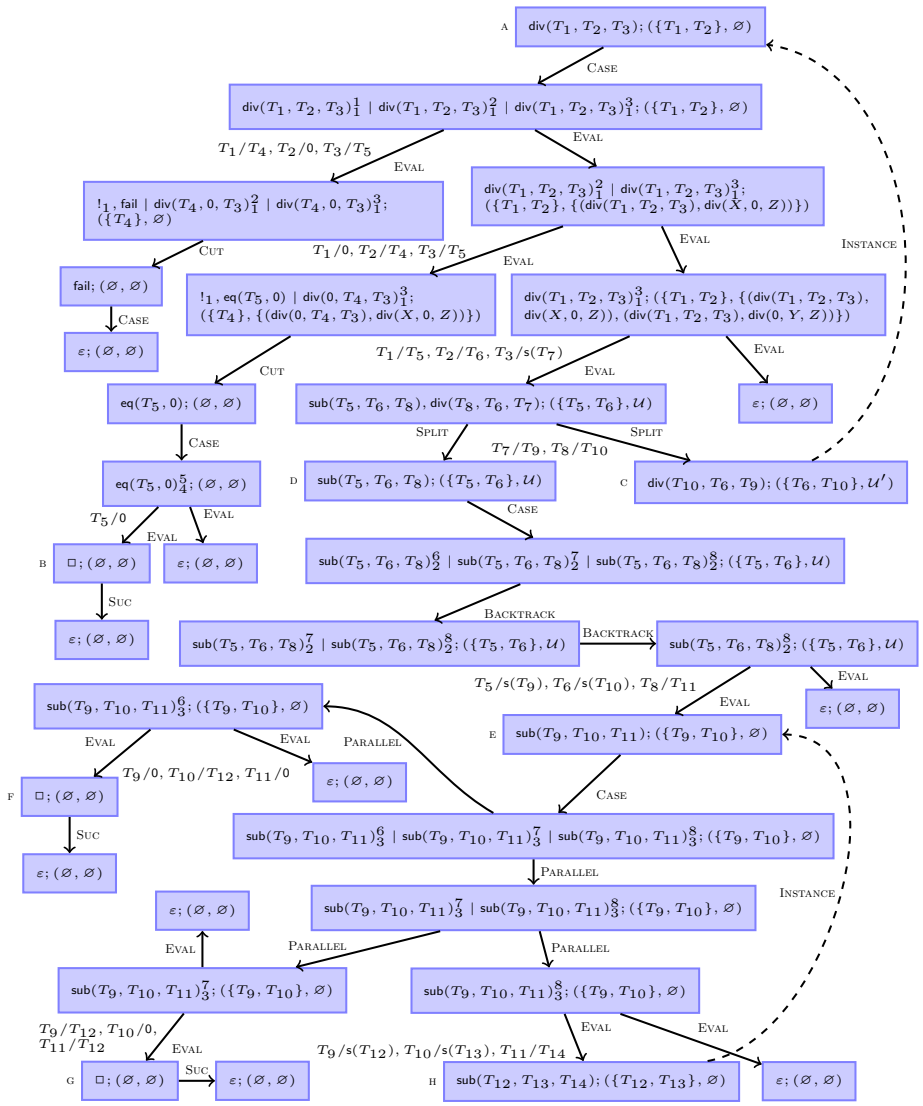
Definition 10 (*Termination Graph*)
A finite graph built from an initial state $(S; KB)$ using Def. 6, 7, 8, and 9 is a *termination graph* iff there is no cycle consisting only of INSTANCE edges and all leaves are of the form $(\varepsilon; KB')$ or $(X, Q \mid S; KB')$ with $X \in \mathcal{V}$. If there are no leaves of the form $(X, Q \mid S; KB')$, then the graph is "*proper*".

We want to generate clauses for the loops in the termination graph and show their termination. Thus, there should be no cycles consisting only of INSTANCE edges, as they would lead to trivially non-terminating clauses. Moreover, the only leaves of the graph may be nodes where no inference rule is applicable anymore (i.e., the graph must be "fully expanded"). Hence, leaves can only be nodes where the state consists only of the empty backtracking sequence ε or where the first goal of the backtracking sequences starts with a variable. For example, the graph at the end of Sect. 4 is a termination graph. Thm. 2 shows that termination graphs can always be obtained automatically.

Theorem 2 (*Existence of Termination Graphs*)
For any program $\mathcal{P}$ and abstract state $(S; KB)$, there exists a termination graph.

Example 10
For the program from Ex. 1 we obtain the termination graph on the next page. Here, $\mathcal{U} = \{(\mathsf{div}(T_5, T_6, T_3), \mathsf{div}(X, \mathsf{0}, Z)), (\mathsf{div}(T_5, T_6, T_3), \mathsf{div}(\mathsf{0}, Y, Z))\}$ results from exploiting the cuts. $\mathcal{U}$ implies that neither T_6 nor T_5 unify with $\mathsf{0}$. Thus, only Clause (8) is applicable to evaluate the state in Node D. This is crucial for termination, because in D, sub's result T_8 is always smaller than sub's input argument T_5 and therefore, div's first argument in Node C is smaller than div's first argument in Node A.

[3] For all proofs, we refer to the appendix.



Remember that our goal is to show termination of the initial state of the graph. Since the graph only has leaves $(\varepsilon; KB)$ that are obviously terminating, by soundness of the inference rules, it remains to prove that there is no state-derivation which would correspond to an infinite traversal of the cycles in the graph. So in our example, we have to show that the INSTANCE edges for div and sub cannot be traversed infinitely often.

We now synthesize a cut-free program from the termination graph. This program has the following property: if there is a state-derivation from a concretization of one state to a concretization of another state which may be crucial for non-termination, then there is a corresponding derivation in the obtained cut-free program.

More precisely, we build clauses for all *clause paths*. For a termination graph G, let $\textsc{Instance}(G)$ denote all nodes of G to which the rule INSTANCE has been applied (i.e., C and H in our example). The sets $\textsc{Split}(G)$ and $\textsc{Suc}(G)$ are defined analogously. For any node n, let $Succ(i, n)$ denote the i-th child of n. Clause paths are paths in the graph that start in the root node, in the successor node of an INSTANCE node, or in the left child of a SPLIT node and that end in a SUC or INSTANCE node or in the left child of an INSTANCE or SPLIT node.

Definition 11 (*Clause Path*)
A path $\pi = n_1 \dots n_k$ in G is a *clause path* iff $k > 1$ and

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

Since we only want finitely many clause paths, they may not traverse INSTANCE edges. Clause paths may also not follow left successors of INSTANCE or SPLIT. Instead, we create new clause paths starting at these nodes. In our example, we have clause paths from A to B, A to C, A to D, D to E, E to F, E to G, and E to H.

To obtain a cut-free logic program, we construct one clause for each clause path $\pi = n_1 \dots n_k$. The head of the new clause corresponds to n_1 where we apply the relevant substitutions between n_1 and n_k. The last body atom corresponds to n_k. The intermediate body atoms correspond to those nodes that are left children of those n_i which are from $\textsc{Split}(G)$. Note that we apply the relevant substitutions between n_i and n_k to the respective intermediate body atom as well.

In our example, the path from A to B is labeled by the substitution $\sigma = \{T_1/\mathsf{0}, T_2/T_4, T_3/\mathsf{0}, T_5/\mathsf{0}\}$. Hence, we obtain the fact $\mathsf{div}_{\text{A}}(T_1, T_2, T_3)\sigma = \mathsf{div}_{\text{A}}(\mathsf{0}, T_4, \mathsf{0})$. We always use a new predicate symbol when translating a node into an atom of a new clause (i.e., div_{A} is fresh). INSTANCE nodes are the only exception. There, we use the same predicate symbol both for the INSTANCE node and its successor.

For the path from A to C, we have the substitution $\sigma' = \{T_1/T_5, T_2/T_6, T_3/\mathsf{s}(T_9), T_7/T_9, T_8/T_{10}\}$. Right children of SPLIT nodes can only be reached if the goal in the left SPLIT-child was successful. So $\mathsf{sub}(T_5, T_6, T_8)\sigma'$ must be derived to $\square$ before the derivation can continue with div. Thus, we obtain the new clause $\mathsf{div}_{\text{A}}(T_5, T_6, \mathsf{s}(T_9)) \leftarrow \mathsf{sub}_{\text{D}}(T_5, T_6, T_{10}), \mathsf{div}_{\text{A}}(T_{10}, T_6, T_9)$. Note that we used the same symbol div_{A} for both occurrences of div as they are linked by an INSTANCE edge.

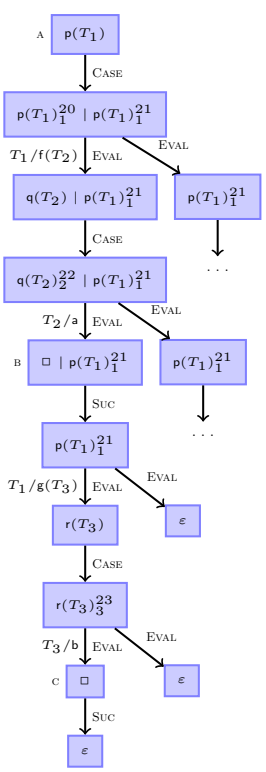
Continuing in this way, we obtain the following logic program for which we have to show termination w.r.t. the set of queries $\{\mathsf{div}_{\text{A}}(t_1, t_2, t_3) \mid t_1, t_2 \text{ are ground}\}$, as specified by the knowledge base in the root node A.

$$
\begin{array}{rcl}
\mathsf{div}_{\text{A}}(\mathsf{0}, T_4, \mathsf{0}). & & \\
\mathsf{div}_{\text{A}}(T_5, T_6, \mathsf{s}(T_9)) & \leftarrow & \mathsf{sub}_{\text{D}}(T_5, T_6, T_{10}), \mathsf{div}_{\text{A}}(T_{10}, T_6, T_9). \\
\mathsf{div}_{\text{A}}(T_5, T_6, \mathsf{s}(T_7)) & \leftarrow & \mathsf{sub}_{\text{D}}(T_5, T_6, T_8). \\
\mathsf{sub}_{\text{D}}(\mathsf{s}(T_9), \mathsf{s}(T_{10}), T_{11}) & \leftarrow & \mathsf{sub}_{\text{E}}(T_9, T_{10}, T_{11}). \\
\mathsf{sub}_{\text{E}}(\mathsf{0}, T_{12}, \mathsf{0}). & & \\
\mathsf{sub}_{\text{E}}(T_{12}, \mathsf{0}, T_{12}). & & \\
\mathsf{sub}_{\text{E}}(\mathsf{s}(T_{12}), \mathsf{s}(T_{13}), T_{14}) & \leftarrow & \mathsf{sub}_{\text{E}}(T_{12}, T_{13}, T_{14}).
\end{array}
\qquad (19)
$$

Virtually all existing methods and tools for proving termination of logic programs succeed on this definite logic program. Hence, by our pre-processing technique, termination of programs with cut like Ex. 1 can be proved automatically.

In general, to convert a node n into an atom, we use a function *Ren*. $Ren(n)$ has the form $p_n(X_1, \ldots, X_n)$ where p_n is a fresh predicate symbol for the node n (except if n is an INSTANCE node) and $X_1, \ldots, X_n$ are all variables in n. This renaming allows us to use different predicate symbols for different nodes. For example, the cut-free logic program above would not terminate if we identified sub_D and sub_E. The reason is that sub_D only succeeds if its first and second argument start with "s". Hence, if the intermediate body atom $\mathsf{sub}_\mathrm{D}(T_5, T_6, T_{10})$ of Clause (19) succeeds, then the "number T_{10}" will always be strictly smaller than the "number T_5". Thus, the first argument in the recursive call of div will be smaller than the first argument in the head of (19). In contrast, sub_E is the ordinary subtraction predicate where the first or second argument can also be $\mathsf{0}$. Finally, *Ren* allows us to represent a whole state by just one atom, even if this state consists of a non-atomic goal or a backtracking sequence with several elements.

The only remaining problem is that paths may contain evaluations for several alternative backtracking goals of the same case analysis. Substitutions that correspond to "earlier" alternatives must not be regarded when instantiating the head of the new clause. The reason is that backtracking undoes the substitutions of previous evaluations. Thus, we collect the substitutions on the path starting with the substitution applied last. Here, we always keep track of the mark d corresponding to the last EVAL node. Substitutions that belong to earlier alternatives of the current case analysis are disregarded when constructing the new cut-free program. These earlier alternatives can be identified easily, since they have marks m with $m \geq d$.



Example 8

Consider the following program and the termination graph for the state $(\mathsf{p}(T_1); (\varnothing, \varnothing))$ on the side. Here, we omitted the knowledge bases to ease readability.

$$\mathsf{p}(\mathsf{f}(X)) \leftarrow \mathsf{q}(X). \quad (20) \qquad \mathsf{q}(\mathsf{a}). \quad (22)$$
$$\mathsf{p}(\mathsf{g}(X)) \leftarrow \mathsf{r}(X). \quad (21) \qquad \mathsf{r}(\mathsf{b}). \quad (23)$$

This graph contains clause paths from A to B and from A to C. For every clause path, we collect the relevant substitutions step by step, starting from the end of the path. So for the first clause path we start with $\{T_2/\mathsf{a}\}$. This substitution results from an EVAL node for the goal $\mathsf{q}(T_2)_2^{22}$ with mark $d = 2$. Hence, for the first clause path we only collect further substitutions that result from EVAL nodes with marks smaller than $d = 2$. Since the next substitution $\{T_1/\mathsf{f}(T_2)\}$ results from an EVAL node with mark 1, we finally obtain $\{T_1/\mathsf{f}(T_2)\} \circ \{T_2/\mathsf{a}\}$ which leads to the fact $\mathsf{p}(\mathsf{f}(\mathsf{a}))$ in the resulting logic program. For the second

clause path from A to C, we start with $\{T_3/\mathsf{b}\}$ which results from an EVAL node with mark $d = 3$. When moving upwards in the tree, the substitution $\{T_1/\mathsf{g}(T_3)\}$ also has to be collected, since it results from an EVAL node with mark 1. Thus, we now set $d = 1$. When moving upwards, we reach further substitutions, but they result from EVAL nodes with marks 2 and 1. These substitutions are not collected, since they correspond to earlier alternatives of this case analysis. Hence, we just obtain the substitution $\{T_1/\mathsf{g}(T_3)\}\circ\{T_3/\mathsf{b}\}$ for the second clause path, which yields the fact $\mathsf{p}(\mathsf{g}(\mathsf{b}))$ in the resulting logic program.

If we disregarded the marks when collecting substitutions, the second clause path would result in $\{T_1/\mathsf{f}(T_2)\}\circ\{T_2/\mathsf{a}\}\circ\{T_1/\mathsf{g}(T_3)\}\circ\{T_3/\mathsf{b}\}$ instead. But then we would get the same fact $\mathsf{p}(\mathsf{f}(\mathsf{a}))$ as from the first clause path. So the new logic program would not simulate all derivations represented in the termination graph.

Now we formally define the cut-free logic program $\mathcal{P}_G$ and the corresponding class of queries $\mathcal{Q}_G$ resulting from a termination graph G. If $\mathcal{P}_G$ is terminating for all queries from $\mathcal{Q}_G$, then the root state of G is terminating w.r.t. the original logic program (possibly containing cuts).

Definition 12 (*Logic Programs and Queries from Termination Graph*)
Let G be a termination graph whose root n is $(p(T_1, ..., T_m), (\{T_{i_1}, ..., T_{i_k}\}, \varnothing))$. We define $\mathcal{P}_G = \bigcup_{\pi \text{ clause path in } G} \mathit{Clause}(\pi)$ and $\mathcal{Q}_G = \{p_n(t_1, ..., t_m) \mid t_{i_1}, ..., t_{i_k} \text{ are ground}\}$. Here, p_n is a new predicate which results from translating the node n into a clause. For a path $\pi = n_1...n_k$, let $\mathit{Clause}(\pi) = \mathit{Ren}(n_1)\sigma_{\pi,\infty} \leftarrow I_\pi, \mathit{Ren}(n_k)$. For $n \in \text{SUC}(G)$, $\mathit{Ren}(n)$ is $\Box$ and for $n \in \text{INSTANCE}(G)$, it is $\mathit{Ren}(\mathit{Succ}(1, n))\mu$ where μ is the substitution associated with the INSTANCE node n. Otherwise, $\mathit{Ren}(n)$ is $p_n(\mathcal{V}(n))$ where p_n is a fresh predicate symbol and $\mathcal{V}(S; KB) = \mathcal{V}(S)$.

Finally, $\sigma_{\pi,d}$ with $d \in \mathbb{N} \cup \{\infty\}$ and I_π are defined as follows. Here for a path $\pi = n_1 \dots n_j$, 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, and the mark m results from the corresponding node $n_{j-1} = ((t, Q)^i_m | S; KB)$.

$$\sigma_{n_1...n_j,d} = \begin{cases} \mathit{id} & \text{if } j = 1 \\ \sigma_{n_1...n_{j-1},d}\ \mu & \text{if } n_{j-1} \in \text{SPLIT}(G),\ n_j = \mathit{Succ}(2, n_{j-1}) \\ \sigma_{n_1...n_{j-1},m}\ \sigma & \text{if } n_{j-1} \in \text{EVAL}(G),\ n_j = \mathit{Succ}(1, n_{j-1}), \text{ and } d > m \\ \sigma_{n_1...n_{j-1},d}\ \sigma|_{\mathcal{G}} & \text{if } n_{j-1} \in \text{EVAL}(G),\ n_j = \mathit{Succ}(1, n_{j-1}), \text{ and } d \leq m \\ \sigma_{n_1...n_{j-1},d} & \text{otherwise} \end{cases}$$

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

So if n_{j-1} is a SPLIT node, then one has to "collect" the corresponding substitution μ when constructing the overall substitution $\sigma_{n_1...n_j,d}$ for the path. If n_{j-1} is an EVAL node for the m-th case analysis and n_j is its left successor, then the construction of $\sigma_{n_1...n_j,d}$ depends on whether we have already collected a corresponding substitution for the current case analysis m. If m is smaller than the mark d for the last case analysis which contributed to the substitution, then the corresponding

substitution σ of the EVAL rule is collected and d is set to m. Otherwise (if $d \leq m$), one only collects the part $\sigma_{|\mathcal{G}}$ of the substitution that concerns those abstract variables that stand for ground terms. The definition of the intermediate body atoms I_π ensures that derivations in $\mathcal{P}_G$ only reach the second child of a SPLIT node if the first child of the SPLIT node could successfully be proved.

Thm. 3 proves the soundness of our approach. So termination of the cut-free program $\mathcal{P}_G$ implies termination of the original program $\mathcal{P}$.

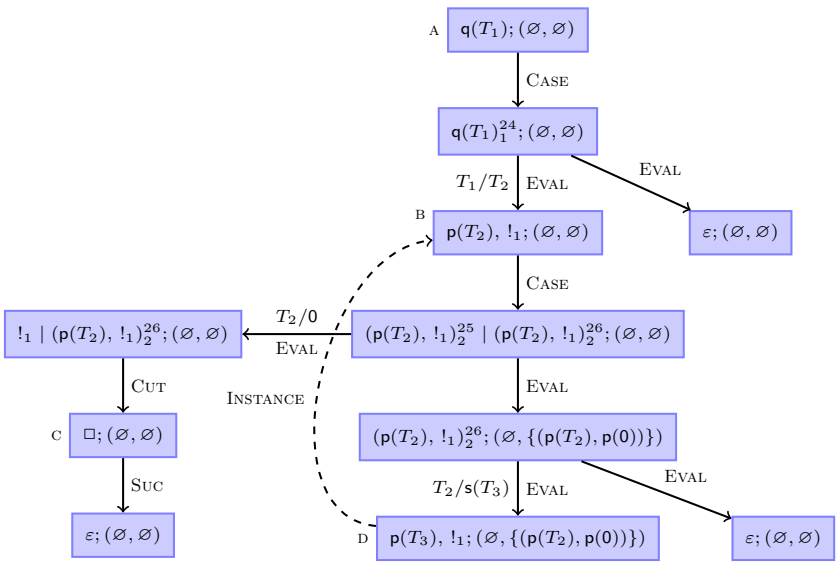
Theorem 3 (*Soundness*)
Let G be a proper termination graph for $\mathcal{P}$ whose root is $(p(T_1, ..., T_m), (\{T_{i_1}, ..., T_{i_k}\}, \varnothing))$. If $\mathcal{P}_G$ terminates for all queries in $\mathcal{Q}_G$, then all concretizations of G's root state have only finite state-derivations. In other words, then all queries from the set $\{p(t_1, \ldots, t_m) \mid t_{i_1}, \ldots, t_{i_k}$ are ground$\}$ terminate w.r.t. $\mathcal{P}$.

The reverse direction of Thm. 3 does not hold. The following example demonstrates that our pre-processing is not termination-preserving.

Example 12
Consider the logic program $\mathcal{P}$ consisting of the following clauses:

$$\mathsf{q}(X) \leftarrow \mathsf{p}(X), !. \quad (24) \qquad \mathsf{p}(\mathsf{0}). \quad (25) \qquad \mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X). \quad (26)$$

We regard the class of all queries $\mathsf{q}(t)$ for arbitrary terms t. The query $\mathsf{q}(t)$ leads to the goal $\mathsf{p}(t), !$. After finitely many resolution steps with Clause (26) that each remove one s-symbol from t, there are two possibilities. First, the derivation can fail because we reach some function symbol in t that is different from $\mathsf{0}$ and s. Second, we reach $\mathsf{0}$ or a variable and, consequently, the cut. Thus, $\mathcal{P}$ terminates for all queries $\mathsf{q}(t)$. We obtain the following termination graph G.



The clause paths are the paths from A to B, from B to C, and from B to D. The

cut-free program $\mathcal{P}_G$ resulting from the termination graph G is

$$\mathsf{q}_{\mathrm{A}}(T_2) \leftarrow \mathsf{p}_{\mathrm{B}}(T_2) \qquad \mathsf{p}_{\mathrm{B}}(\mathsf{0}). \qquad \mathsf{p}_{\mathrm{B}}(\mathsf{s}(T_3)) \leftarrow \mathsf{p}_{\mathrm{B}}(T_3).$$

The class of queries $\mathcal{Q}_G$ that we have to analyze for termination are all queries of the form $\mathsf{q}_{\mathrm{A}}(t)$. But, for instance, the query $\mathsf{q}_{\mathrm{A}}(X)$ does not terminate in the program $\mathcal{P}_G$. The problem here is that the unification information $\mathcal{U}$ is disregarded when constructing the clauses of the cut-free program. This information states that the second p_{B}-clause can only be used if the argument of p_{B} does not unify with $\mathsf{0}$. If one took this information into account, then the resulting program $\mathcal{P}_G$ would terminate. However, expressing this information in terms of program clauses would lead to much more complex resulting programs which would often make their termination proof much harder or even impossible with current automated tools.

6 Experiments and Conclusions

We introduced a pre-processing method to eliminate cuts. Afterwards, any technique for proving universal termination of logic programming can be applied. Thus, termination of logic programs with cuts can now be analyzed automatically.

We implemented this pre-processing in our tool AProVE (Giesl et al. 2006) and performed extensive experiments which show that now we can indeed prove termination of typical logic programs with cut fully automatically. The implementation is not only successful for programs like Ex. 1, but also for programs using operators like *negation as failure* or *if then else* which can be expressed using cuts. While AProVE was already one of the most powerful termination tools for definite logic programs (Schneider-Kamp et al. 2009), our pre-processing method strictly increases its power. For our experiments, we used the *Termination Problem Database* (TPDB) of the annual *International Termination Competition*.[4] Since up to now, no tool had special support for cuts, the previous versions of the TPDB did not contain any programs with cuts. Therefore, we took existing cut-free examples from the TPDB and added cuts in a natural way. In this way, we extended the TPDB by 104 typical programs with cuts (directory `LP/CUT`). Of these, 10 are known to be non-terminating. Up to now, termination tools treated cuts by simply ignoring them and by trying to prove termination of the program that results from removing the cuts. This is a sensible approach, since cuts are not always needed for termination. Indeed, a version of AProVE that ignores cuts and does not use our pre-processing can show termination of 10 of the 94 potentially terminating examples. Other existing termination tools would not yield much better results, since AProVE is already the most powerful tool for definite logic programming (as shown by the experiments in (Schneider-Kamp et al. 2009)) and since most of the remaining 84 examples do not terminate anymore if one removes the cut. In contrast, with our new pre-processing, AProVE proves termination of 78 examples (i.e., 83% of the potentially terminating examples). This shows that our contributions are crucial for termination analysis of logic programs with cuts. Nevertheless, there is of course

[4] `http://termination-portal.org/wiki/Termination_Competition`

room for further improvements (e.g., one could develop alternative techniques to generate cut-free clauses from the termination graph in order to improve the performance on examples which encode existential termination). To experiment with our implementation and for further details, we refer to

`http://aprove.informatik.rwth-aachen.de/eval/Cut/`

Acknowledgements. We thank the referees for many helpful remarks.

References

Andrews, J. H. 2003. The witness properties and the semantics of the Prolog cut. *TPLP 3,* 1, 1–59.

Apt, K. R. 1997. *From Logic Programming to Prolog.* Prentice Hall, London.

Billaud, M. 1990. Simple operational and denotational semantics for Prolog with cut. *Theor. Comp. Sc. 71,* 2, 193–208.

Bruynooghe, M., Codish, M., Gallagher, J. P., Genaim, S., and Vanhoof, W. 2007. Termination analysis of logic programs through combination of type-based norms. *ACM TOPLAS 29,* 2.

Codish, M., Lagoon, V., and Stuckey, P. J. 2005. Testing for termination with monotonicity constraints. In *ICLP '05.* LNCS 3668. 326–340.

Cousot, P. and Cousot, R. 1992. Abstract interpretation and application to logic programs. *J. Log. Prog. 13,* 2-3, 103–179.

De Schreye, D. and Decorte, S. 1994. Termination of logic programs: The never-ending story. *J. Log. Prog. 19,20,* 199–260.

de Vink, E. P. 1989. Comparative semantics for Prolog with cut. *Sci. Comp. Prog. 13,* 1, 237–264.

Deransart, P., Ed-Dbali, A., and Cervoni, L. 1996. *Prolog: The Standard.* Springer, New York.

Filé, G. and Rossi, S. 1993. Static analysis of Prolog with cut. In *LPAR '93.* LNAI 698. 134–145.

Giesl, J., Schneider-Kamp, P., and Thiemann, R. 2006. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *IJCAR '06.* LNAI 4130. 281–286.

Giesl, J., Swiderski, S., Schneider-Kamp, P., and Thiemann, R. 2006. Automated termination analysis for Haskell: From term rewriting to programming languages. In *RTA '06.* LNCS 4098. 297–312.

Howe, J. M. and King, A. 2003. Efficient groundness analysis in Prolog. *TPLP 3,* 1, 95–124.

Kulas, M. and Beierle, C. 2000. Defining standard Prolog in rewriting logic. In *WRLA '00.* ENTCS 36.

Le Charlier, B., Rossi, S., and Van Hentenryck, P. 1994. An abstract interpretation framework which accurately handles Prolog search-rule and the cut. In *ILPS '94.* MIT Press, 157–171.

Marchiori, M. 1996. Proving existential termination of normal logic programs. In *AMAST '96.* LNCS 1101. 375–390.

Mesnard, F. and Serebrenik, A. 2007. Recurrence with affine level mappings is P-time decidable for CLP(R). *TPLP 8,* 1, 111–119.

Mogensen, T. Æ. 1996. A semantics-based determinacy analysis for Prolog with cut. In *Ershov Memorial Conference*. LNCS 1181. 374–385.

Nguyen, M. T., De Schreye, D., Giesl, J., and Schneider-Kamp, P. 2010. Polytool: Polynomial interpretations as a basis for termination analysis of logic programs. *TPLP*. To appear.

Schneider-Kamp, P., Giesl, J., Serebrenik, A., and Thiemann, R. 2009. Automated termination proofs for logic programs by term rewriting. *ACM TOCL 11,* 1.

Serebrenik, A. and De Schreye, D. 2005. On termination of meta-programs. *TPLP 5,* 3, 355–390.

Sørensen, M. H. and Glück, R. 1995. An algorithm of generalization in positive supercompilation. In *ILPS '95*. MIT Press, 465–479.

Spoto, F. and Levi, G. 1998. Abstract interpretation of Prolog programs. In *AMAST '98*. LNCS 1548. 455–470.

Spoto, F. 2000. Operational and goal-independent denotational semantics for Prolog with cut. *J. Log. Prog. 42,* 1, 1–46.

Ströder, T. 2010. Towards termination analysis of real Prolog programs. Diploma Thesis, RWTH Aachen. `http://aprove.informatik.rwth-aachen.de/eval/Cut/`.

Appendix A Proof of Theorem 1

We prove Theorem 1 by showing the Lemmas 1-5. In the Lemmas 1 and 2, to prove soundness we show the stronger property that the adapted abstract rules fulfill the *simulation property*, i.e., that each concrete state-derivation step is captured by one of the adapted abstract rules.

Lemma 1 (*Soundness of* Suc, Fail, Cut, *and* Case)
The rules Suc, Fail, Cut, and Case from Def. 6 are sound.

Proof
For Suc we need to show that for all concretizations $\square \mid S\gamma \in \mathcal{C}on(\square \mid S; KB)$ we reach a state $S\gamma \in \mathcal{C}on(S; KB)$ by the concrete Suc rule. Assume that we have a concrete state-derivation from $\square \mid S\gamma \in \mathcal{C}on(\square \mid S; KB)$. The only rule from Def. 2 that is applicable is the concrete Suc rule. Thus, this derivation must start with a step from $\square \mid S\gamma$ to $S\gamma$. As γ is a concretization w.r.t. KB, we have that $S\gamma \in \mathcal{C}on(S; KB)$, which concludes our proof for Suc.

Likewise, for the first Cut rule we have to show that for all concretizations $!_m, Q\gamma \mid S\gamma \mid ?_m \mid S'\gamma \in \mathcal{C}on(!_m, Q \mid S \mid ?_m \mid S'; KB)$ where S contains no $?_m$ we reach a state $Q\gamma \mid ?_m \mid S'\gamma \in \mathcal{C}on(Q \mid ?_m \mid S'; KB)$ by the first concrete Cut rule. To this end, note that the first step in the derivation of $!_m, Q\gamma \mid S\gamma \mid ?_m \mid S'\gamma$ has to be an application of the first concrete Cut rule resulting in $Q\gamma \mid ?_m \mid S'\gamma$. As KB remains unchanged, we immediately obtain $Q\gamma \mid ?_m \mid S'\gamma \in \mathcal{C}on(Q \mid ?_m \mid S'; KB)$. For the second Cut rule, assume that we have a state-derivation from $!_m, Q\gamma \mid S\gamma$ and S does not contain $?_m$. Hence, $S\gamma$ also does not contain $?_m$. Then, the only applicable concrete rule is the second Cut rule. We obtain $Q\gamma \in \mathcal{C}on(Q; KB)$ as we did not change the knowledge base.

For the Case rule, assume that we have a state-derivation from $t\gamma, Q\gamma \mid S\gamma \in \mathcal{C}on(t, Q \mid S; KB)$. The only applicable concrete rule is Case, which results in $(t\gamma, Q\gamma)^{i_1}_m \mid \ldots \mid (t\gamma, Q\gamma)^{i_k}_m \mid ?_m \mid S\gamma$. As we did not change KB, this concrete state is an element of $\mathcal{C}on((t, Q)^{i_1}_m \mid \ldots \mid (t, Q)^{i_k}_m \mid ?_m \mid S; KB)$.

For Fail, assume that we have a state-derivation from $?_m \mid S\gamma \in \mathcal{C}on(?_m \mid S; KB)$. The only applicable concrete rule is Fail, which results in $S\gamma$. As we did not change KB, this concrete state is an element of $\mathcal{C}on(S; KB)$. $\square$

Lemma 2 (*Soundness of* Backtrack *and* Eval)
The rules Backtrack and Eval from Def. 7 are sound. Additionally, for Eval, for every concretization γ w.r.t. $(\mathcal{G}, \mathcal{U})$ with $mgu(t\gamma, H_i) = \sigma'$ there is a concretization γ' w.r.t. $(\mathcal{G}', \mathcal{U}\sigma|_{\mathcal{G}})$ such that $\gamma\sigma' = \sigma\gamma'$, $\gamma = \sigma|_{\mathcal{G}}\gamma'$ and $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})}$.

Proof
For Backtrack, assume that there is a state-derivation from $(t\gamma, Q\gamma)^i_m \mid S\gamma \in \mathcal{C}on((t, Q)^i_m \mid S; (\mathcal{G}, \mathcal{U}))$. By definition, there is no concretization γ' such that $t\gamma' \sim H_i$. In particular, we have that $t\gamma \not\sim H_i$ and, therefore, the only applicable concrete

rule is BACKTRACK, which results in $S\gamma$. As we did not change KB, this concrete state is an element of $\mathcal{C}on(S;KB)$.

For EVAL, assume that we have a concrete state-derivation from $(t\gamma, Q\gamma)^i_m \mid S\gamma \in \mathcal{C}on((t,Q)^i_m \mid S;(\mathcal{G},\mathcal{U}))$. There are two cases depending on whether $t\gamma$ and H_i unify.

First, if $t\gamma$ does not unify with H_i, the only applicable concrete rule is BACKTRACK and we obtain $S\gamma$. From $t\gamma \not\sim H_i$, $\mathcal{V}(H_i) \subseteq \mathcal{N}$, and $Dom(\gamma) = \mathcal{A}$, we know that $H_i\gamma = H_i$ and, therefore, $t\gamma \not\sim H_i\gamma$ and γ is a concretization w.r.t. $(\mathcal{G},\mathcal{U} \cup \{(t,H_i)\})$. Thus, $S\gamma \in \mathcal{C}on(S;(\mathcal{G},\mathcal{U} \cup \{(t,H_i)\}))$.

Second, if $t\gamma \sim H_i$, the only applicable concrete rule is EVAL. From $H_i\gamma = H_i$ we know that $t\gamma \sim H_i\gamma$ and thus t also unifies with H_i. Let $mgu(t\gamma, H_i) = \sigma''$. Then using $H_i\gamma = H_i$ shows that $\gamma\sigma''$ is a unifier of t and H_i. As $mgu(t,H_i) = \sigma$ there is a substitution σ''' such that $\gamma\sigma'' = \sigma\sigma'''$. W.l.o.g., we demand that $\mathcal{V}(Range(\sigma''))$ are fresh variables from $\mathcal{N}$.

By application of the concrete EVAL rule we obtain $B'_i\sigma'', Q\gamma\sigma'' \mid S\gamma$ where $B'_i = B_i[!/!_m]$. We are, thus, left to show that $B'_i\sigma'', Q\gamma\sigma'' \mid S\gamma \in \mathcal{C}on(B'_i\sigma, Q\sigma \mid S\sigma|_{\mathcal{G}};(\mathcal{G}',\mathcal{U}\sigma|_{\mathcal{G}}))$, i.e., that there is a concretization γ' w.r.t. $(\mathcal{G}',\mathcal{U}\sigma|_{\mathcal{G}})$ such that $B'_i\sigma'' = B'_i\sigma\gamma'$, $Q\gamma\sigma'' = Q\sigma\gamma'$, and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

We define $\gamma'(T) = \sigma'''(T)$ for $T \in \mathcal{A}(Range(\sigma))$ and $\gamma'(T) = \gamma(T)$ for $T \in \mathcal{A} \setminus \mathcal{A}(Range(\sigma))$. As $Range(\sigma)$ contains only fresh variables, we clearly have that $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})}$.

We start by showing that γ' is a concretization w.r.t. $(\mathcal{G}',\mathcal{U}\sigma|_{\mathcal{G}})$, i.e., $Dom(\gamma') = \mathcal{A}$, $\mathcal{V}(Range(\gamma')) \subseteq \mathcal{N}$, $\mathcal{V}(Range(\gamma'|_{\mathcal{G}'})) = \varnothing$, and $\bigwedge_{(r,r')\in\mathcal{U}\sigma|_{\mathcal{G}}} r\gamma' \not\sim r'\gamma'$.

As γ' is only defined for $\mathcal{A}$, we trivially have $Dom(\gamma') \subseteq \mathcal{A}$. For the other direction, assume that there is some $T \in \mathcal{A}$ such that $\gamma'(T) = T$. As γ is a concretization we know by the definition of γ' that then $T = \gamma'(T) = \sigma'''(T)$ and $T \in \mathcal{A}(Range(\sigma))$. But then T occurs in $\mathcal{A}(Range(\sigma\sigma'''))$ which by the definition of σ'' is the same as $\mathcal{A}(Range(\gamma\sigma''))$. This yields the desired contradiction as $\mathcal{A}(Range(\gamma\sigma'')) = \varnothing$ since γ is a concretization and σ'' also does not introduce variables of $\mathcal{A}$.

To show that $\mathcal{V}(Range(\gamma')) \subseteq \mathcal{N}$, we perform a case analysis w.r.t. the partition $\mathcal{A} = \mathcal{A}(Range(\sigma)) \uplus (\mathcal{A} \setminus (\mathcal{A}(Range(\sigma)))$. For the case that $T \in \mathcal{A}(Range(\sigma))$ we have $\mathcal{A}(T\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(T\sigma''') \overset{T\notin Dom(\sigma)}{=} \mathcal{A}(T\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{A}(T\gamma\sigma'') \overset{\mathcal{V}(Range(\sigma''))\subseteq\mathcal{N}}{=} \mathcal{A}(T\gamma) \overset{\mathcal{V}(Range(\gamma))\subseteq\mathcal{N}}{=} \varnothing$. For $T \in \mathcal{A} \setminus (\mathcal{A}(Range(\sigma)))$ we have $\mathcal{A}(T\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(T\gamma) \overset{\mathcal{V}(Range(\gamma))\subseteq\mathcal{N}}{=} \varnothing$.

Now we show $\mathcal{V}(Range(\gamma'|_{\mathcal{G}'})) = \varnothing$. For $T \in \mathcal{G}'$ we conclude $T \in \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ and thus, we have $\mathcal{V}(T\gamma') = \mathcal{V}(T\sigma''')$. For all $T' \in Dom(\sigma|_{\mathcal{G}})$, $\mathcal{V}(T'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{V}(T'\gamma\sigma'') \overset{T'\in\mathcal{G}}{=} \varnothing$. Thus, $\mathcal{V}(T\gamma') = \varnothing$.

Finally, we have $\bigwedge_{(r,r')\in\mathcal{U}\sigma|_{\mathcal{G}}} r\gamma' \not\sim r'\gamma' = \bigwedge_{(s,s')\in\mathcal{U}} s\sigma|_{\mathcal{G}}\gamma' \not\sim s'\sigma|_{\mathcal{G}}\gamma' = \bigwedge_{(s,s')\in\mathcal{U}} s\gamma \not\sim s'\gamma$ as $T\sigma|_{\mathcal{G}}\gamma' = T\gamma$ for all abstract variables $T \in \mathcal{A}(\mathcal{U})$ by definition of γ'. To see this, consider the partition $\mathcal{A}(\mathcal{U}) = (\mathcal{A}(\mathcal{U}) \setminus Dom(\sigma|_{\mathcal{G}})) \uplus (\mathcal{A}(\mathcal{U}) \cap Dom(\sigma|_{\mathcal{G}}))$. If $T \in \mathcal{A}(\mathcal{U}) \setminus Dom(\sigma|_{\mathcal{G}})$ we have $T\gamma \overset{Def.\gamma'}{=} T\gamma' \overset{T\notin Dom(\sigma|_{\mathcal{G}})}{=} T\sigma|_{\mathcal{G}}\gamma'$. If $T \in \mathcal{A}(\mathcal{U}) \cap Dom(\sigma|_{\mathcal{G}})$ we have $T\gamma \overset{T\in\mathcal{G}}{=} T\gamma\sigma'' \overset{Def.\sigma'''}{=} T\sigma\sigma''' \overset{T\in\mathcal{G}}{=} T\sigma|_{\mathcal{G}}\sigma''' \overset{Def.\gamma'}{=} T\sigma|_{\mathcal{G}}\gamma'$.

Now, we are left to show that $B'_i\sigma'' = B'_i\sigma\gamma'$, $Q\gamma\sigma'' = Q\sigma\gamma'$, and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

Then $\gamma\sigma' = \sigma\gamma'$ and $\gamma = \sigma|_{\mathcal{G}}\gamma'$ follows from the fact that we only defined γ' differently from γ for variables in the range of σ.

For S there are two cases according to the partition $\mathcal{A}(S) = (\mathcal{A}(S) \setminus Dom(\sigma|_{\mathcal{G}})) \uplus (\mathcal{A}(S) \cap Dom(\sigma|_{\mathcal{G}}))$. Analogous to the analysis for $\mathcal{A}(\mathcal{U})$ above, we have $T\gamma = T\sigma|_{\mathcal{G}}\gamma'$ for both cases. With $Dom(\gamma) = \mathcal{A}$, $Dom(\gamma') = \mathcal{A}$, and $Dom(\sigma|_{\mathcal{G}}) \subseteq \mathcal{G} \subseteq \mathcal{A}$ we obtain $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

For B'_i we have $\mathcal{V}(B'_i) = \mathcal{N}(B'_i)$ and for all $x \in \mathcal{N}(B'_i)$ we obtain: $x\sigma'' \overset{Dom(\gamma)=\mathcal{A}}{=} x\gamma\sigma'' \overset{Def.\sigma'''}{=} x\sigma\sigma''' \overset{x\in Dom(\sigma)\wedge Def.\gamma'}{=} x\sigma\gamma'$. Thus, $B'_i\sigma'' = B'_i\sigma\gamma'$.

Now, for Q let $T \in \mathcal{V}(Q)$. As $T \in Dom(\sigma)$, we have $T\gamma\sigma'' \overset{Def.\sigma'''}{=} T\sigma\sigma''' \overset{Def.\gamma'}{=} T\sigma\gamma'$. Thus, we obtain $Q\gamma\sigma'' = Q\sigma\gamma'$. □

For the remaining Lemmas 3, 4, and 5 we cannot use the simulation property to prove soundness. Instead, we use a proof by contrapositive, i.e., we show that if any concretization of an abstract state has an infinite concrete state-derivation, then some successor of the abstract state has an infinite concrete state-derivation.

Lemma 3 (*Soundness of* Instance)
The rule Instance from Def. 8 is sound. Additionally, for every concretization γ w.r.t. $(\mathcal{G},\mathcal{U})$ there is a concretization γ' w.r.t. $(\mathcal{G}',\mathcal{U}')$ such that $S\gamma = S'\gamma'\mu|_{\mathcal{N}}$.

Proof
Assume we have an infinite concrete state-derivation starting from $S\gamma \in \mathcal{C}on(S;(\mathcal{G},\mathcal{U}))$. We show that there is a substitution γ' such that $S'\gamma' \in \mathcal{C}on(S';(\mathcal{G}',\mathcal{U}'))$ and $S'\gamma'$ has an infinite concrete state-derivation.

As $\mu|_{\mathcal{N}}$ is a variable renaming, there is a μ^{-1} such that $\mu|_{\mathcal{N}}\mu^{-1} = \mu^{-1}\mu|_{\mathcal{N}} = id$. Let $\gamma' = \mu\gamma\mu^{-1}$. Clearly, as $S'\mu = S$ and μ^{-1} is a variable renaming, $S'\gamma' = S\gamma\mu^{-1}$ has an infinite concrete state-derivation. Additionally, we have that $S'\gamma'\mu|_{\mathcal{N}} = S\gamma\mu^{-1}\mu|_{\mathcal{N}} = S\gamma$. We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}',\mathcal{U}')$.

For $x \in \mathcal{N}$ we have $x\mu \in \mathcal{N}$ and, thus, $x\gamma' \overset{Def.\gamma'}{=} x\mu\gamma\mu^{-1} \overset{x\mu\in\mathcal{N}}{=} x\mu\mu^{-1} \overset{Def.\mu^{-1}}{=} x$, i.e., $Dom(\gamma') = \mathcal{A}$. From the definition of μ^{-1} we obtain $\mathcal{V}(Range(\mu^{-1})) \subseteq \mathcal{N}$. Together with $\mathcal{V}(Range(\gamma)) \subseteq \mathcal{N}$ and $Dom(\gamma) = \mathcal{A}$ we have $\mathcal{V}(Range(\gamma')) = \mathcal{V}(Range(\mu\gamma\mu^{-1}) \subseteq \mathcal{N}$.

We know that for all $T \in \mathcal{G}'$, $\mathcal{V}(T\mu) \subseteq \mathcal{G}$. Further, as γ is a concretization w.r.t. $(\mathcal{G},\mathcal{U})$ we know that for all $T \in \mathcal{G}$, $\mathcal{V}(Range(\gamma|_{\mathcal{G}})) = \varnothing$. Thus, for all $T \in \mathcal{G}'$, we have $\mathcal{V}(T\gamma') \overset{Def.\gamma'}{=} \mathcal{V}(T\mu\gamma\mu^{-1}) = \mathcal{V}(T\mu\gamma) = \varnothing$ and, therefore, $\mathcal{V}(Range(\gamma'|_{\mathcal{G}'})) = \varnothing$.

Finally, from $\bigwedge_{(s,s')\in\mathcal{U}} s\gamma \not\sim s'\gamma$ and $\mathcal{U}'\mu \subseteq \mathcal{U}$, we know that $\bigwedge_{(s,s')\in\mathcal{U}'\mu} s\gamma \not\sim s'\gamma$ which is equivalent to $\bigwedge_{(s,s')\in\mathcal{U}'} s\mu\gamma \not\sim s'\mu\gamma$. As μ^{-1} is a variable renaming, trivially $\bigwedge_{(s,s')\in\mathcal{U}'} s\mu\gamma\mu^{-1} \not\sim s'\mu\gamma\mu^{-1}$ and, consequently, $\bigwedge_{(s,s')\in\mathcal{U}'} s\gamma' \not\sim s'\gamma'$.

This concludes our proof as $\gamma' = \mu\gamma\mu^{-1}$ satisfies all conditions of a concretization w.r.t. $(\mathcal{G}',\mathcal{U}')$. □

Lemma 4 (*Soundness of* PARALLEL)
The rule PARALLEL from Def. 8 is sound. Additionally, for every concretization γ w.r.t. KB we have that if the concrete state-derivation for $S\gamma \mid S'\gamma$ reaches a state of the form "$\ldots \mid S'\gamma'$" for some concretization γ' w.r.t. KB, then $\gamma' = \gamma$.

Proof
Assume that $S\gamma \mid S'\gamma \in \mathcal{C}on(S \mid S'; KB)$ has an infinite concrete state-derivation. There are three cases. If $S\gamma$ has an infinite concrete state-derivation, we immediately have that $S\gamma \in \mathcal{C}on(S; KB)$ has an infinite concrete state-derivation. If $S\gamma$ does not have an infinite concrete state-derivation and, after finitely many steps, we reach the state $S'\gamma$, we have that $S'\gamma \in \mathcal{C}on(S'; KB)$ has an infinite concrete state-derivation. Finally, if $S\gamma$ has no infinite concrete state-derivation, but we do not reach $S'\gamma$, we know that S' must be of the form $S'' \mid ?_m \mid S'''$ with $S'' \neq \varepsilon$ and in the concrete state-derivation of $S\gamma \mid S'\gamma$ we apply the CUT rule to $!_m, Q \mid S''''\gamma \mid S''\gamma \mid ?_m \mid S'''\gamma$, i.e., $m \in AC(S)$. As $S\gamma \mid S'\gamma$ has an infinite concrete state-derivation, we get $S''' \neq \varepsilon$. But $S'' \neq \varepsilon \neq S'''$ implies $m \in AM(S')$. Thus we have a contradiction to $AC(S) \cap AM(S') = \varnothing$. In particular, we do not reach a state of the form "$\ldots \mid S'\gamma'$" for some concretization γ' w.r.t. KB such that $\gamma' \neq \gamma$. □

Lemma 5 (*Soundness of* SPLIT)
The rule SPLIT from Def. 9 is sound. Additionally, for every concretization γ w.r.t. $(\mathcal{G},\mathcal{U})$ and for every answer substitution μ' of a successful concrete state-derivation for $t\gamma$, there is a concretization γ' w.r.t. $(\mathcal{G}',\mathcal{U}\mu)$ such that $\gamma\mu' = \mu\gamma'$ and $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})}$.

Proof
Assume that $t\gamma, Q\gamma \in \mathcal{C}on(t, Q; (\mathcal{G},\mathcal{U}))$ has an infinite concrete state-derivation. There are two cases. If $t\gamma$ has an infinite concrete state-derivation, we immediately have that $t\gamma \in \mathcal{C}on(t; (\mathcal{G},\mathcal{U}))$ has an infinite concrete state-derivation. If $t\gamma$ does not have an infinite concrete state-derivation and we did not reach a state of the form $Q\gamma\mu' \mid S'\gamma$ for some answer substitution μ' and state S', we would reach the state ε, which contradicts our assumption that $t\gamma, Q\gamma$ has an infinite concrete state-derivation. Therefore, if $t\gamma$ does not have an infinite concrete state-derivation, we reach states of the form $Q\gamma\mu' \mid S'\gamma$ for answer substitutions μ' and states S'. If all $Q\gamma\mu'$ did not have an infinite concrete state-derivation, this would contradict our assumption that $t\gamma, Q\gamma$ has an infinite concrete state-derivation. Thus, there must be a state $Q\gamma\mu'$ that has an infinite concrete state-derivation. We now show that there is a concretization γ' such that $\gamma\mu' = \mu\gamma'$ for all answer substitutions μ' corresponding to a successful concrete state-derivation of $t\gamma$. Then, in particular, we have an infinite concrete state-derivation from $Q\mu\gamma' \in \mathcal{C}on(Q\mu; (\mathcal{G}',\mathcal{U}\mu))$.

The answer substitution μ' can potentially instantiate any non-ground term in $Q\gamma$. We define γ' in such a way that $\gamma\mu' = \mu\gamma'$ and $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)}$. This is always possible because all variables in $Range(\mu)$ are fresh. Then, clearly, $Q\gamma\mu' = Q\mu\gamma'$ and $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)}$. We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}',\mathcal{U}\mu)$. As we only need to define γ' for abstract variables,

clearly $Dom(\gamma') = \mathcal{A}$. From $\mathcal{V}(Range(\mu')) \subseteq \mathcal{N}$ and $\mathcal{V}(Range(\gamma)) \subseteq \mathcal{N}$ we know that $\mathcal{V}(Range(\gamma')) \subseteq \mathcal{N}$. We perform a case analysis based on the partition $\mathcal{G}' = \mathcal{G} \uplus (ApproxGnd(t,\mu) \setminus \mathcal{G})$. For $T \in \mathcal{G}$ we have effectively defined $T\gamma' = T\gamma$ and, thus, $\mathcal{V}(T\gamma') = \mathcal{V}(T\gamma) = \varnothing$. For $T \in ApproxGnd(t,\mu) \setminus \mathcal{G}$ by definition of $ApproxGnd$ and equality of $\gamma\mu'$ and $\mu\gamma'$ we know that $T\gamma'$ is a ground term, i.e., $\mathcal{V}(T\gamma') = \varnothing$. For all $(s,s') \in \mathcal{U}$ we have $s\gamma \not\sim s'\gamma$ and, consequently $s\gamma\mu' \not\sim s'\gamma\mu'$. But from $s\gamma\mu' = s\mu\gamma'$ and $s'\gamma\mu' = s'\mu\gamma'$ we get $s\mu\gamma' \not\sim s'\mu\gamma'$. Thus, for all $(s'',s''') \in \mathcal{U}\mu$, we have $s''\gamma' \not\sim s'''\gamma'$. □

Proof of Theorem 1

The soundness of the inference rules from Definitions 6 and 7 follows from Lemmas 1 and 2, respectively. The soundness of the rules from Definition 8 follows from Lemmas 3 and 5. Finally, soundness of the inference rule from Definition 9 follows from Lemma 5. □

Appendix B Proof of Theorem 2

Proof of Theorem 2

Let $\mathcal{P}$ be a program using the function symbols $p_1/k_1, \ldots, p_n/k_n \in \Sigma$. W.l.o.g. we consider a state consisting of only one unlabeled goal with only one term $p_1(t_1, \ldots, t_{k_1})$. We apply the INSTANCE rule to generalize this state to a new state such that the term has pairwise different abstract variables as arguments and an empty knowledge base. Thus, we obtain the state $p(T_1, \ldots, T_{k_1}); (\varnothing, \varnothing)$. Then we apply the CASE rule and obtain a state having a number of labeled goals $(p_1(T_1, \ldots, T_{k_1}))^i_m$ and a question mark $?_m$. We use the PARALLEL rule to separate all elements of this state and obtain a number of states consisting of only one labeled goal $(p_1(T_1, \ldots, T_{k_1}))^i_m$ or the question mark $?_m$. We use the FAIL rule to evaluate $?_m$ to the empty state ε. Then we apply BACKTRACK to evaluate as many states as possible to the empty state as well. For the remaining states we use the EVAL rule and obtain a number of states having only one unlabeled goal with a number of terms $t_1, \ldots, t_l$. We use the SPLIT rule repeatedly to separate all terms in these goals and obtain a number of states consisting of only one unlabeled goal with only one term t_j. For all such states with $t_j = !_m$ we apply the CUT rule to obtain the empty state. For the remaining states with $t_j = p_i(t'_1, \ldots, t'_{k_i})$ for some $i \in \{1, \ldots, n\}$, we use the INSTANCE rule to a (possibly already existing) state $p_i(T'_1, \ldots, T'_{k_i}); (\varnothing, \varnothing)$ in the graph with only one unlabeled goal having only one term which has the same root symbol and pairwise different variables. The remaining leaves are of a form like our first state, but with a different root symbol, or consist of a variable only. As we only have a finite number of different function symbols in the program, after performing the above procedure for at most n times, the construction terminates and we obtain a termination graph for $\mathcal{P}$ as all leaves are of the form $\varepsilon; (\varnothing, \varnothing)$ or $X; (\varnothing, \varnothing)$ for some $X \in \mathcal{V}$. □

Appendix C Proof of Theorem 3

Definition 13 (*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 $S'_i = ?_{f(m)}$
- $S_i = \Box$ implies $S_i = S'_i$
- $S_i = Q$ implies $S'_i = Q', Q''$ for a list 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 list 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'.

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

Definition 14 (*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 state-derivation we would have to fork on Parallel nodes, but as we will be interested in the relevant parts of the concrete state-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 6 (*Success Tree for Concrete State-Derivations in Termination Graph*)
Let $S\gamma \in \mathcal{C}on(S;KB)$ with $S;KB = n \in V$ for a termination graph $G = (V,E)$ and there is a concrete state-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 $n' = S';KB'$, $S'\gamma' \in \mathcal{C}on(S';KB')$, $S'\gamma'\rho$ 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 state-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 $S_i\gamma_i \in \mathcal{C}on(S_i;KB_i)$
- for all leaves (i,v_i,p_i) of π with $i \neq k$ we have $v_i \in \textsc{Suc}(G)$
- for all (i,v_i,p_i) with more than one successor in π, 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, 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 state-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 $S'\gamma'' \in \mathcal{C}on(S';KB')$ and $S\gamma = S'\gamma''\mu|_{\mathcal{N}}$. As $\mu|_{\mathcal{N}}$ is a variable renaming we conclude that the concrete state-derivation from $S\gamma$ to a state extension of S'' can be completely simulated by a corresponding concrete state-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 state-derivation from $S\gamma$ to a state extension of S'' then there also is an i-th state S'_i in the concrete state-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 state-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 state-derivation reaches a state extension of $S_2\gamma$. If the concrete state-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 state-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 state-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 state-derivation reaches a state extension of $\Box$ from $t\gamma$. As this concrete state-derivation is shorter than the one of $(t, Q)\gamma$ we obtain a node $n'' \in \text{SUC}(G)$ 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 state-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 state-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 $S\gamma = S'' \in \mathcal{C}on(S; KB)$. 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 state-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 state-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 6 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 state-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 6 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 state-derivation.

- For EVAL we have $S = (t, Q)^i_j \mid S_r$ and the concrete state-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 6 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 state-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 6 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 7 (*Single Concretization*)
Given a path $\pi = n_1 \dots n_k$ with $n_j \notin \text{INSTANCE}(G)$ for all $j \in \{1, \dots, k-1\}$ and a concrete state-derivation such that there are variable renamings $\rho_1, \dots, \rho_k$ and concretizations $\gamma_1, \dots, \gamma_k$ 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 state-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 $n_1 \in \text{SPLIT}(G)$ 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 a variable renaming ρ and a concretization γ' w.r.t. KB_j for all $j \in \{2, \dots, k\}$ 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 Lemma 5 and the fact that the concrete state-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 $n_1 = (t, Q)^c_m \mid S; KB$ 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 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 Lemma 2 and the fact that the concrete state-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 $(t, Q)^c_m\gamma_1\rho_1 \mid S\gamma_1\rho_1$ 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 8 (*Answer Substitutions are Instances of Clause Path Substitutions*)
Given a path $\pi = n_1 \ldots n_k$ with $n_j \notin \textsc{Instance}(G)$ for all $j \in \{1,\ldots,k-1\}$ and a concrete state-derivation such that there is a variable renaming ρ and a concretization γ 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 state-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 $n_2 = \mathit{Succ}(2, n_1)$, 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 $\sigma_{n_2 \ldots n_k,\infty}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete state-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 state-derivation from a state extension of $(t, Q)\gamma\rho$ to a state extension of $Q\mu\gamma\rho$ we know by Lemma 5 that $\gamma\rho\mu' = \mu\gamma\rho$. Therefore, we have $\gamma\rho\delta = \gamma\rho\mu'\delta'' = \mu\gamma\rho\delta'' = \mu\sigma_{n_2 \ldots n_k,\infty}\gamma\rho = \sigma_{\pi,\infty}\gamma\rho$. 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 $n_2 = \mathit{Succ}(1, n_1)$, 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 $\sigma_{n_2 \ldots n_k,\infty}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete state-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 state-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 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 $j \in \{2, \ldots, k-1\}$ with a scope less or equal than m. Since the scopes are ascendingly ordered, we know that the concrete state-derivation did not backtrack the substitution σ'. Hence, we obtain $\gamma\rho\delta = \gamma\rho\sigma'\delta'' = \sigma\gamma\rho\delta'' = \sigma\sigma_{n_2 \ldots n_k,\infty}\gamma\rho = \sigma_{\pi,\infty}\gamma\rho$. 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 m. Since the scopes are ascendingly ordered, we know that the concrete state-derivation did backtrack the substitution σ' and we have the same answer substitution δ'' for the complete concrete state-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 state-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 state-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 $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

□

Lemma 9 (*Simulation of Concrete State-Derivations using $\mathcal{P}(G)$*)
If γ_0 is a concretization w.r.t. KB_0 such that there is a concrete state-derivation from $S_0\gamma_0\rho_0$ to a state extension of $S_k\gamma_k\rho_k$, where γ_i and ρ_i are taken of the success tree for the concrete state-derivation according to Lemma 6, and if $S_0; KB_0 \in Succ(1, \textsc{Instance}(G) \cup \textsc{Split}(G))$ or $S_0; KB_0$ is the root of G, and if $S_k \in \textsc{Suc}(G) \cup \textsc{Instance}(G)$, then $Ren(n_0)\gamma_0\rho_0 \vdash^*_{\mathcal{P}(G)} Ren(n_k)\gamma_k\rho_k$. Moreover, if $k > 0$ then there is at least one $\vdash_{\mathcal{P}(G)}$-step in this derivation.

Proof
As we have built the success tree w.r.t. Lemma 6 we know that there is a tree path $\pi = (0, n_0, none), (1, n_1, p_1), \ldots, (k, n_k, p_k)$ with the following properties:

- $n_0 = S_0; KB_0$
- for all $i \in \{0, \ldots, k\}$ there are concretizations γ_i and variable renamings ρ_i on $\mathcal{N}$ such that the concrete state-derivation reaches a state extension of $S_i\gamma_i\rho_i$ where $n_i = S_i; KB_i$
- for all (i, n_i, p_i) with more than one successor in π, we have $n_i \in \textsc{Split}(G)$
- for all (i, n_i, p_i) with $n_i \in \textsc{Split}(G)$ and only one successor (j, n_j, i) in π, we have $n_j = Succ(1, n_i)$
- for all leaves (i, n_i, p_i) of π where $i \neq k$ we have $n_i \in \textsc{Suc}(G)$
- $n_k \in \textsc{Suc}(G) \cup \textsc{Instance}(G)$

To prove the lemma we first define some auxiliary notions: $\mathsf{right}(\pi)$ is the rightmost (linear) path $n_0 \ldots n_k$ in π. Moreover, for a linear path $n_0 \ldots n_k$, $\mathsf{cl}(n_0 \ldots n_k)$ is the prefix $n_0 \ldots n_{k'}$ of that path such that $n_0 \ldots n_{k'}$ is nearly a clause path. It must satisfy all conditions of the definition of a clause path except that the length may be 1 and that there is no condition on the initial node. Throughout this proof we write $\vdash$ instead of $\vdash_{\mathcal{P}(G)}$. Moreover, we often write S_k instead of $S_k; KB_k$.

To prove the lemma we show the following essential property.

$$Ren(n_0)\gamma_0\rho_0 \;\vdash_{Ren(n_0)\sigma_{\mathsf{cl}(\mathsf{right}(\pi)),\infty} \leftarrow I_{\mathsf{cl}(\mathsf{right}(\pi))}, Ren(n_{k'})} {}^{*}\; Ren(n_k)\gamma_k\rho_k. \tag{C1}$$

Note from (C1) we can directly conclude the lemma: if $k = 0$ then we do not even have to use (C1). Otherwise, if $k > 0$ then by the requirement of the lemma that $S_0; KB_0$ is the root node or it is in $Succ(1, \textsc{Split}(G) \cup \textsc{Instance}(G))$ we know that $\mathsf{cl}(\mathsf{right}(\pi))$ is a clause path. Thus, the clause $Ren(n_0)\sigma_{\mathsf{cl}(\mathsf{right}(\pi)),\infty} \leftarrow I_{\mathsf{cl}(\mathsf{right}(\pi))}, Ren(n_{k'})$ is from $\mathcal{P}_\mathcal{G}$ and the lemma follows directly from (C1).

Now, we prove (C1) by induction on π where we do not demand that n_0 is the root of G or that $n_0 \in Succ(1, \textsc{Split}(G) \cup \textsc{Instance}(G))$.

First, if $k = 0$ then we obtain $Ren(n_0)\gamma_0\rho_0 \vdash_{Ren(n_0) \leftarrow Ren(n_0)} Ren(n_0)\gamma_0\rho_0$. Note, that $\mathsf{cl}(\mathsf{right}(\pi)) = n_0$, and hence, the generated clause is indeed $Ren(n_0) \leftarrow Ren(n_0)$.

Otherwise, π contains at least two nodes. Then $\mathsf{right}(\pi) = n_0 n_1 \ldots n_k$. Let $\mathsf{cl}(\mathsf{right}(\pi)) = n_0 \ldots n_{k'} =: \pi'$.

If $n_0 \in \textsc{Parallel}(G)$, $n_0 = S_1 \mid S_2$, and S_2 is not reached in the concrete state-derivation then by the construction of π we know that $n_1 = Succ(1, n_0)$, $\gamma_0 = \gamma_1$, $\rho_0 = \rho_1$, and the concrete state-derivation is of the form $(S_1 \mid S_2)\gamma_0\rho_0 \vdash^* S_k\gamma_k\rho_k \mid$

S' where already $S_1\gamma_0 \vdash^* S_k\gamma_k\rho_k; S''$ for some state S'' (which is shorter than S'). Hence, for the subtree of π with n_1 as root we can apply the induction hypothesis to obtain $Ren(n_1)\gamma_1\rho_1 \vdash_{Ren(n_1)\sigma_{n_1\dots n_{k'},\infty} \leftarrow I_{n_1\dots n_{k'}}, Ren(n_{k'})} \vdash^* Ren(n_k)\gamma_k\rho_k$. But since $n_1 \in \textsc{Parallel}(G)$ we know that $\sigma_{\pi',\infty} = \sigma_{n_1\dots n_{k'},\infty}$ and $I_{\pi'} = I_{n_1\dots n_{k'}}$. Therefore, $Ren(n_1)\gamma_0\rho_0 \vdash_{Ren(n_1)\sigma_{\pi',\infty} \leftarrow I_{\pi'}, Ren(n_{k'})} \vdash^* S_k\gamma_k\rho_k$. Moreover, we can easily replace $Ren(n_1)$ by $Ren(n_0)$ in the above derivation and are done.

In the other case for n_0 being a Parallel-node we know that $n_1 = Succ(2, n_0)$ and the concrete state-derivation is of the form $(S_1 \mid S_2)\gamma_0\rho_0 \vdash^* S'' \vdash S_k\gamma_k\rho_k \mid S'$ where S'' is a state-extension of $S_2\gamma_2\rho_2$, and where already $S_2\gamma_2\rho_2$ can be derived to a state-extension of $S_k\gamma_k\rho_k$ by the same steps. Then one can again apply the induction hypothesis and continue as in the previous case.

The cases $n_0 \in \textsc{Suc}(G) \cup \textsc{Fail}(G) \cup \textsc{Cut}(G) \cup \textsc{Backtrack}(G) \cup \textsc{Case}(G)$ are handled in the same way as for the parallel node. If $n_0 \in \textsc{Split}(G)$ and $n_1 = Succ(1, n_0)$ or if $n_0 \in \textsc{Eval}(G)$ and $n_1 = Succ(2, n_0)$ the reasoning is also similar.

So, let us now consider $n_0 \in \textsc{Split}(G)$, $n_0 = t, Q$, and $n_1 = Succ(2, n_0) = Q\mu$. Let $n'' = Succ(1, n_0) = t$ and π'' be the corresponding subtree of π where for n'' the substitutions γ'' and ρ'' are γ_0 and ρ_0, respectively. We conclude that the concrete state-derivation is of the form $(t, Q)\gamma_0\rho_0 \vdash^* (\Box, Q\mu, Q' \mid S'')\gamma_1\rho_1 \vdash (Q\mu, Q' \mid S'')\gamma_1\rho_1 \vdash^* S_k\gamma_k\rho_k \mid S'$ where already $Q\mu\gamma_1\rho_1$ can be derived to a state extension of $S_k\gamma_k\rho_k$. Moreover, the rightmost node n''' in π'' is a success-node. Note, that since $n'' \in Succ(1, \textsc{Split}(G))$, we know that $\mathsf{cl}(\mathsf{right}(\pi''))$ is a proper clause path. Hence, the corresponding clause is contained in $\mathcal{P}(G)$. Thus, by the induction hypothesis we obtain $Ren(n'')\gamma_0\rho_0 \vdash^* Ren(n''')\gamma'''\rho''' = \Box$ and since ρ_0 is a variable renaming we also have $Ren(n'')\gamma_0 \vdash^* \Box$. Moreover, for the answer substitution δ corresponding to the evaluation of n_0 the induction hypothesis also yields $Ren(n_1)\gamma_1\rho_1 \vdash_{Ren(n_1)\sigma_{n_1\dots n_{k'},\infty} \leftarrow I_{n_1\dots n_{k'}}, Ren(n_{k'})} \vdash^* Ren(n_k)\gamma_k\rho_k$, i.e., $(I_{n_1\dots n_{k'}}, Ren(n_{k'}))\delta = (I_{n_1\dots n_{k'}}, Ren(n_{k'})) \vdash^* Ren(n_k)\gamma_k\rho_k$, because of Lemma 8. Looking at the definition of I, we see that $I_{\pi'} = Ren(n'')\sigma_{\pi'}I_{n_1\dots n_{k'}}$. Moreover, by Lemma 7 and Lemma 8 we know that $\sigma_{\pi',\infty}\gamma_0 = \gamma_0\delta$ As in the range of γ_0 only variables of $\mathcal{N}$ occur, w.l.o.g. δ only operates on $\mathcal{N}$. We obtain

$$
\begin{aligned}
& Ren(n_0)\gamma_0\rho_0 \\
& \vdash_{Ren(n_0)\sigma_{\pi',\infty} \leftarrow I_{\pi'}, Ren(n_{k'})} (I_{\pi'}, Ren(n_{k'}))\gamma_0\rho_0\delta \\
& = (I_{\pi'}, Ren(n_{k'}))\gamma_0\rho_0 \\
& = Ren(n'')\sigma_{\pi',\infty}\gamma_0\rho_0, (I_{n_1\dots n_{k'}}, Ren(n_{k'}))\gamma_0\rho_0 \\
& = Ren(n'')\gamma_0\rho_0\delta, (I_{n_1\dots n_{k'}}, Ren(n_{k'}))\gamma_0\rho_0 \\
\text{(ind.)} & \vdash^* (I_{n_1\dots n_{k'}}, Ren(n_{k'}))\gamma_0\rho_0\delta \\
& = (I_{n_1\dots n_{k'}}, Ren(n_{k'}))\gamma_0\rho_0 \\
\text{(ind.)} & \vdash^* Ren(n_k)\gamma_k\rho_k
\end{aligned}
$$

For the next case we assume $n_0 \in \textsc{Instance}(G)$. Let n_1 be the first successor from n_0 with $n_1 \notin \textsc{Instance}(G)$. Then $n_1 = S$, $n_0 = S\mu$ where μ is the composition of all matching substitutions along the path from n_0 to n_1. This path must be finite as we do not have cycles consisting only of Instance edges. Furthermore,

we know that $Ren(n_0) = Ren(n_1)\mu$. Let π'' be the subtree of π with root n_1. Then $\mathsf{cl}(\mathsf{right}(\pi''))$ is a proper clause path. Hence, by the induction hypothesis we obtain $Ren(n_1)\gamma_1\rho_1 \vdash^* Ren(n_k)\gamma_k\rho_k$. Moreover, $\pi' = \mathsf{cl}(\mathsf{right}(\pi)) = n_0 = n_{k'}$. Hence, the clause $Ren(n_0)\sigma_{\pi',\infty} \leftarrow I_{\pi'}, Ren(n_{k'})$ is $Ren(n_0) \leftarrow Ren(n_0)$ as $I_{\pi'} = \Box$ and $\sigma_{\pi',\infty} = id$. We also know $S\mu\gamma_0\rho_0 = n_0\gamma_0\rho_0 = n_1\gamma_1\rho_1 = S\gamma_1\rho_1$ and hence, $x\mu\gamma_0\rho_0 = x\gamma_1\rho_1$ for all $x \in \mathcal{V}(S) = \mathcal{V}(Ren(n_1))$. Putting all information together yields the desired derivation.

$$\begin{aligned} & Ren(n_0)\gamma_0\rho_0 \\ & \vdash_{Ren(n_0)\leftarrow Ren(n_0)} Ren(n_0)\gamma_0\rho_0 \\ & = Ren(n_1)\mu\gamma_0\rho_0 \\ & = Ren(n_1)\gamma_1\rho_1 \\ (\text{ind.}) & \vdash^* Ren(n_k)\gamma_k\rho_k \end{aligned}$$

Finally, we have to consider the case that $n_0 \in \textsc{Eval}(G)$ and $n_1 = Succ(1, n_0)$. Then $n_0 = (t, Q)^i_m \mid S$, $n_1 = B'_i\sigma, Q\sigma \mid S\sigma|_{\mathcal{G}}$, $\pi' = n_0 n_1 \dots n_{k'}$, and $I_{\pi'} = I_{n_1\dots n_{k'}}$. Moreover, we obtain $Ren(n_1)\gamma_1\rho_1 \vdash_{Ren(n_1)\sigma_{n_1\dots n_{k'},\infty}\leftarrow I_{n_1\dots n_{k'}}, Ren(n_{k'})}{}^* Ren(n_k)\gamma_k\rho_k$ from the induction hypothesis. Furthermore, $\sigma_{\pi',\infty} = \sigma_{n_0 n_1\dots n_{k'},\infty}$. By Lemma 7 and Lemma 8 we obtain w.l.o.g. that $\gamma_0 = \gamma_1 = \gamma_{k'}$, $\rho_0 = \rho_1 = \rho_{k'}$ and where $\sigma_{\pi',\infty}\gamma_0\rho_0 = \gamma_0\delta_0\rho_0$ for the answer substitution δ_0 of the concrete state-derivation $n_0\gamma_0\rho_0$ to the state extension of $n_{k'}\gamma_0\rho_0$. Moreover, $S_k\gamma_0\rho_0\delta_0 = S_k\gamma_0\rho_0$. From Lemma 2 we know that $\gamma_0\rho_0 = \sigma|_{\mathcal{G}}\gamma_0\rho_0$ and $\gamma_0\rho_0\sigma' = \sigma\gamma_0\rho_0$ where $\sigma' = mgu(t\gamma_0\rho_0, H_i)$. Furthermore, by $\sigma_{\pi',\infty}\gamma_0\rho_0 = \gamma_0\rho_0\delta_0$ we obtain the unifier $\delta_0\gamma_0\rho_0$ of $\sigma_{\pi',\infty}$ and $\gamma_0\rho_0$. Hence,

$$\begin{aligned} & Ren(n_0)\gamma_0\rho_0 \\ & \vdash_{Ren(n_0)\sigma_{\pi',\infty}\leftarrow I_{\pi'}, Ren(n_{k'})} (I_{\pi'}, Ren(n_{k'}))\gamma_0\rho_0\delta_0 \\ & = (I_{\pi'}, Ren(n_{k'}))\gamma_0\rho_0 \\ & = (I_{n_1\dots n_{k'}}, Ren(n_{k'}))\gamma_0\rho_0 \\ (\text{ind.}) & \vdash^* Ren(n_k)\gamma_k\rho_k \end{aligned}$$

□

Proof of Theorem 3

Assume that $\mathcal{P}_G$ is terminating for all queries in $\mathcal{Q}_G$, but that there is a concretization $S\gamma \in \mathcal{C}on(S; KB)$ from the root node $n_0 = (S; KB)$ of G that has an infinite concrete state-derivation starting from $S\gamma$. Then, according to Lemma 6 there is an infinite tree path π_{tree} where the rightmost path $\pi = n_0, n_1, n_2, \dots$ in π_{tree} is an infinite sequence of clause paths $\pi_0, \pi_1, \pi_2, \dots$ and there are indices $l_0, l_1, l_2, \dots$ such that $\pi_m = n_{l_m}, \dots$.

According to Lemma 9, we have $Ren(n_{l_m})\gamma_{l_m}\rho_{l_m} \vdash^+_{\mathcal{P}_G} Ren(n_{l_{m+1}})\gamma_{l_{m+1}}\rho_{l_{m+1}}$ for all m. Thus, $Ren(n_{l_0})\gamma_{l_0}\rho_{l_0} = Ren(n_0)\gamma_0\rho_0 = Ren(n_0)\gamma_0$ has an infinite derivation w.r.t. $\mathcal{P}_G$. As $S\gamma_0 \in \mathcal{C}on(n_0)$, $\mathcal{P}_G$ is not terminating w.r.t. all queries from $\mathcal{Q}_G$. This contradicts our initial assumption and, thus, 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:** (*e-mail:* `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 vom 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

* These reports are only available as a printed version.

Please contact (*e-mail:* `biblio@informatik.rwth-aachen.de`) to obtain copies.