RWTH Aachen

Department of Computer Science
Technical Report

Automated Complexity Analysis for Prolog by Term Rewriting

Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2012-05

RWTH Aachen · Department of Computer Science · March 2012

Automated Complexity Analysis for Prolog by Term Rewriting∗

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

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

CARSTEN FUHS
Dept. of Computer Science, University College London, United Kingdom

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

For term rewrite systems (TRSs), a huge number of automated termination analysis techniques have been developed during the last decades, and by automated transformations of Prolog programs to TRSs, these techniques can also be used to prove termination of Prolog programs. Very recently, techniques for automated termination analysis of TRSs have been adapted to prove asymptotic upper bounds for the runtime complexity of TRSs automatically. In this paper, we present an automated transformation from Prolog programs to TRSs such that the runtime of the resulting TRS is an asymptotic upper bound for the runtime of the original Prolog program (where the runtime of a Prolog program is measured by the number of unification attempts). Thus, techniques for complexity analysis of TRSs can now also be applied to prove upper complexity bounds for Prolog programs.

Our experiments show that this transformational approach indeed yields more precise bounds than existing direct approaches for automated complexity analysis of Prolog. Moreover, it is also applicable to a larger class of Prolog programs such as non-well-moded programs or programs using built-in predicates like, e.g., cuts.

KEYWORDS: complexity analysis, automated reasoning, logic programs, term rewriting

1 Introduction

Automated complexity analysis of term rewrite systems has recently gained a lot of attention (see, e.g., (Avanzini et al. 2008; Avanzini and Moser 2009; Bonfante et al. 2001; Hirokawa and Moser 2008; Marion and Péchoux 2008; Noschinski et al. 2011; Waldmann 2010; Zankl and Korp 2010)). Most of these complexity analysis techniques were obtained by adapting existing approaches for termination analysis of TRSs. Indeed, complexity analysis can be seen as a refinement of termination analysis: Instead of only asking whether a program will eventually halt, one asks

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

how many steps it will take before the program halts. This view is also apparent in the competition on automated complexity analysis of TRSs, which takes place as part of the annual *International Termination Competition*[1] since 2008, and where most of the competing tools are built on the basis of a termination analyzer.

In the area of *termination analysis*, there exist several *transformational approaches* which permit the use of techniques for automated termination proofs of TRSs also for termination analysis of logic programs. To this end, logic programs are automatically transformed into TRSs in a non-termination preserving way (see, e.g., (Ohlebusch 2001)). In fact, this transformational approach for termination analysis of logic programs turned out to be more powerful than techniques to analyze termination of logic programs directly (Schneider-Kamp et al. 2009).

In this paper, we develop a similar *transformational approach* for *complexity analysis*. While there already exists some work on direct complexity analysis for logic programs (e.g., (Debray and Lin 1993; López-García et al. 2010)[2]), these approaches are restricted to well-moded logic programs. By making complexity analysis of TRSs applicable to logic programs as well, we obtain an approach for automated complexity analysis of Prolog which is applicable to a much wider class of programs (including non-well-moded and non-definite programs).[3] Moreover, as shown by extensive experiments, the implementation of our approach in the tool AProVE (Giesl et al. 2006) is far more powerful than the previous direct approaches.

We introduce the required notations, the considered operational semantics, and the notion of complexity for Prolog programs in Sect. 2. In Sect. 3 we show that existing transformations from logic programs to TRSs, which were originally developed for termination analysis, cannot be directly used for complexity, as they do not preserve asymptotic upper complexity bounds. The reason is that backtracking in the logic program is replaced by non-deterministic choice in the TRS.

Thus, we propose a new transformation based on a *derivation graph* which represents all possible executions of a logic program. This is similar to our approach for termination analysis in (Schneider-Kamp et al. 2010; Ströder et al. 2010) which goes beyond definite logic programs. In this way, the transformation is also applicable to Prolog programs using built-in predicates like cuts. We explain derivation graphs in Sect. 4. Then in Sect. 5, we present a method to obtain TRSs from such graphs which have at least the same complexity as the original Prolog program. To this end, we also developed a new criterion for determinacy analysis of Prolog (Hill and King 1997). In Sect. 6, we compare our approach to the existing direct ones empirically.

2 Preliminaries

Let Σ be a set of function symbols. Each $\mathsf{f} \in \Sigma$ has an arity $n \in \mathbb{N}$ denoted f/n. We always assume that Σ contains at least one constant symbol. Moreover, let $\mathcal{V}$ be a countably infinite set of variables. The set of *terms* $\mathcal{T}(\Sigma, \mathcal{V})$ is the least set where

[1] See `http://www.termination-portal.org/wiki/Termination_Competition`

[2] Moreover, there also exist approaches to infer *lower* complexity bounds for logic programs, (e.g., (Debray et al. 1997; King et al. 1997)), whereas our approach can only infer *upper* bounds.

[3] However, our implementation currently does not treat built-in integer arithmetic, whereas (Debray and Lin 1993; López-García et al. 2010) can handle linear arithmetic constraints.

$\mathcal{V} \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ and where $\mathsf{f}(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$ for all $\mathsf{f}/n \in \Sigma$ and $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$. $\mathcal{V}(t)$ denotes the set resp. the sequence of variables in a term t. For a term $t = \mathsf{f}(t_1, \ldots, t_n)$, we have $root(t) = \mathsf{f}/n$. A *position* $pos \in \mathbb{N}^*$ in a term t addresses a subterm $t|_{pos}$ of t. We denote the empty word (and thereby the top position) by ε. The term $t[s]_{pos}$ results from replacing the subterm $t|_{pos}$ at position pos in t by the term s. So $t|_\varepsilon = t$ and $t[s]_\varepsilon = s$. For $pos = i\ pos'$, $i \in \mathbb{N}$, and $t = \mathsf{f}(t_1, \ldots, t_n)$, we have $t|_{pos} = t|_{i\ pos'} = t_i|_{pos'}$ and $t[s]_{pos} = t[s]_{i\ pos'} = \mathsf{f}(t_1, \ldots, t_i[s]_{pos'}, \ldots, t_n)$.

For the basics of term rewriting, see, e.g., (Baader and Nipkow 1998). A *term rewrite system (TRS)* $\mathcal{R}$ is a finite set of pairs of terms $\ell \to r$ (called rules) where $\ell \notin \mathcal{V}$ and $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$. The rewrite relation $s \to_\mathcal{R} t$ for two terms s and t holds iff there is an $\ell \to r \in \mathcal{R}$, a position pos, and a substitution σ such that $\ell\sigma = s|_{pos}$ and $t = s[r\sigma]_{pos}$. The rewrite step is *innermost* (denoted $s \xrightarrow{i}_\mathcal{R} t$) iff no proper subterm of $\ell\sigma$ can be rewritten. The *defined symbols* of a TRS $\mathcal{R}$ are $\Sigma_d = \{root(\ell) \mid \ell \to r \in \mathcal{R}\}$, i.e., these are the function symbols that can be "evaluated".

Different notions of complexity have been proposed for TRSs. In this paper, we focus on *innermost runtime complexity* (Hirokawa and Moser 2008), which corresponds to the notion of complexity used for programming languages. Here, one only considers rewrite sequences starting with *basic* terms $\mathsf{f}(t_1, \ldots, t_n)$, where $\mathsf{f} \in \Sigma_d$ and $t_1, \ldots, t_n$ do not contain symbols from Σ_d. The *innermost runtime complexity function* $irc_\mathcal{R}$ maps any $n \in \mathbb{N}$ to the length of the longest sequence of $\xrightarrow{i}_\mathcal{R}$-steps starting with a basic term t where $|t| \leq n$. Here, $|t|$ is the number of variables and function symbols occurring in t. To measure the complexity of a TRS $\mathcal{R}$, we determine the asymptotic size of $irc_\mathcal{R}$, i.e., we say that $\mathcal{R}$ has linear complexity iff $irc_\mathcal{R}(n) \in \mathcal{O}(n)$, quadratic complexity iff $irc_\mathcal{R}(n) \in \mathcal{O}(n^2)$, etc.

See, e.g., (Apt 1997) for the basics of logic programming. As in the ISO standard for Prolog (ISO/IEC 13211-1 1995), we do not distinguish between predicate and function symbols. A *query* is a sequence of terms, where $\square$ denotes the empty query. A *clause* is a pair h :- B where the *head* h is a term and the *body* B is a query. If B is empty, then one writes just "h" instead of "h :- $\square$". A *Prolog program* $\mathcal{P}$ is a finite sequence of clauses. In this paper, we consider unification with occurs check.[4] If s and t have no *mgu* σ, we write $mgu(s,t) = fail$. $Slice_\mathcal{P}(\mathsf{p}(t_1, ..., t_n))$ is the sequence of all program clauses "h :- B" from $\mathcal{P}$ where $root(h) = \mathsf{p}/n$.

We consider the operational semantics in (Ströder et al. 2011) which is equivalent to the semantics in (ISO/IEC 13211-1 1995). A *state* has the form $\langle G_1 \mid \ldots \mid G_n \rangle$ where $G_1 \mid \ldots \mid G_n$ is a sequence of goals. Essentially, a *goal* is just a *query*, i.e., a sequence of terms. In addition, a goal can also be labeled by a clause c, where the goal $(t_1, \ldots, t_k)^c$ indicates that the next resolution step has to be performed with clause c. Intuitively, a state $\langle G_1 \mid \ldots \mid G_n \rangle$ means that we currently have to solve the goal G_1, but that $G_2, \ldots, G_n$ are the next goals to solve when backtracking.[5] The *initial state* for a query $(t_1, \ldots, t_k)$ is $\langle (t_1, \ldots, t_k) \rangle$, i.e., this state contains just a single goal. The operational semantics can be defined by a set of inference rules on these states.

[4] Our method could be extended to unification without occurs check, but we left this as future work since the complexity of most programs does not depend on the occurs check.

[5] We omit answer substitutions for simplicity, since they do not contribute to the complexity.

$$\frac{\square \mid S}{S}\ (\textsc{Suc}) \qquad \frac{(t,Q) \mid S}{(t,Q)^{c_1} \mid \cdots \mid (t,Q)^{c_a} \mid S}\ (\textsc{Case})\ \text{ if } \mathit{Slice}_{\mathcal{P}}(t) = (c_1,\ldots,c_a)$$

$$\frac{(t,Q)^{h\,:\!-\,B} \mid S}{(B\sigma, Q\sigma) \mid S}\ (\textsc{Eval})\ \text{ if } \mathit{mgu}(t,h) = \sigma \qquad \frac{(t,Q)^{h\,:\!-\,B} \mid S}{S}\ (\textsc{Backtrack})\ \text{ if } \mathit{mgu}(t,h) = \mathit{fail}$$

Fig. 1. Inference Rules for the Subset of Definite Logic Programs

In Fig. 1, we show the inference rules for the part of Prolog which defines definite logic programming. Here, S denotes a (possibly empty) sequence of goals. The set of all inference rules for full Prolog can be found in (Ströder et al. 2011). Since each state contains all backtracking goals, our semantics is *linear* (i.e., a *derivation* with these rules is just a sequence of states and not a search tree as in the classic Prolog semantics). As outlined in (Ströder et al. 2011), this makes our semantics particularly well suited for termination and complexity analysis.

For a Prolog program $\mathcal{P}$ and a query Q, we consider the length of the longest derivation starting in the initial state for Q. As shown in (Ströder et al. 2011), this length is equal to the number of unification attempts when traversing the whole SLD tree according to the semantics of (ISO/IEC 13211-1 1995), up to a constant factor.

Thus, we use the length of this longest derivation to measure the complexity of Prolog programs.[6] We consider classes of atomic queries which are described by a $\mathsf{p} \in \Sigma$ and a *moding function* $m : \Sigma \times \mathbb{N} \to \{in, out\}$. So m determines which arguments of a symbol are considered to be input. The corresponding class of queries is $\mathcal{Q}^{\mathsf{p}}_m = \{\mathsf{p}(t_1,\ldots,t_n) \mid \mathcal{V}(t_i) = \varnothing \text{ for all } i \text{ with } m(\mathsf{p},i) = in\}$. For a moding function m, and any term $\mathsf{p}(t_1,\ldots,t_n)$, its *moded size* is $|\mathsf{p}(t_1,\ldots,t_n)|_m = \Sigma_{i\in\{1,\ldots,n\}:m(\mathsf{p},i)=in}\,|t_i|$. Thus, for a program $\mathcal{P}$ and a class of queries $\mathcal{Q}^{\mathsf{p}}_m$, the Prolog *runtime complexity function* $prc_{\mathcal{P},\mathcal{Q}^{\mathsf{p}}_m}$ maps any $n \in \mathbb{N}$ to the length of the longest derivation starting with the initial state for some query $Q \in \mathcal{Q}^{\mathsf{p}}_m$ with $|Q|_m \le n$. For a program $\mathcal{P}$ and a class of queries $\mathcal{Q}^{\mathsf{p}}_m$, our aim is to generate a TRS $\mathcal{R}$ such that asymptotically, $irc_{\mathcal{R}}(n)$ is an upper bound of $prc_{\mathcal{P},\mathcal{Q}^{\mathsf{p}}_m}(n)$.

3 Direct Transformation

Consider the following program `sublist.pl` from the *Termination Problem Data Base (TPDB)*[7] with the class of queries $\mathcal{Q}^{\mathsf{sublist}}_m$. Here m is a moding function with $m(\mathsf{sublist},1) = out$ and $m(\mathsf{sublist},2) = in$.

```
(1)  app([],Ys,Ys).
(2)  app(.(X,Xs),Ys,.(X,Zs)) :- app(Xs,Ys,Zs).
(3)  sublist(X,Y)             :- app(P,U,Y), app(V,X,P).
```

This program computes (by backtracking) all sublists of a given list. Its complexity

[6] In contrast, (Debray and Lin 1993; López-García et al. 2010) use the number of resolution steps to measure complexity. As long as we do not consider dynamic built-in predicates like assert/1, these measures are asymptotically equivalent, as the number of failing unification attempts is bounded by a constant factor (i.e., by the number of clauses in the program).

[7] This is the collection of examples used in the annual International Termination Competition.

w.r.t. $\mathcal{Q}_m^{\mathsf{sublist}}$ is quadratic since the first call to app takes a linear number of unification attempts and produces also a linear number of solutions. The second call to app again needs linear time, but due to backtracking, it is called linearly often.

We now show that the classic transformation from (well-moded) logic programs to TRSs (see, e.g., (Ohlebusch 2001)) cannot be used for complexity analysis.[8] Note that the example program is well moded if m is extended to app by defining $m(\mathsf{app}, 1) = m(\mathsf{app}, 2) = \mathit{out}$ and $m(\mathsf{app}, 3) = \mathit{in}$. For each predicate p, the transformation introduces two new function symbols p^{in} and $\mathsf{p}^{\mathit{out}}$. Let "$\mathsf{p}(\vec{s}, \vec{t})$" denote that $\vec{s}$ and $\vec{t}$ are the sequences of terms on p's *in*- and *out*-positions.

- For each fact $\mathsf{p}(\vec{s}, \vec{t})$, the TRS contains the rule $\mathsf{p}^{\mathit{in}}(\vec{s}) \to \mathsf{p}^{\mathit{out}}(\vec{t})$.
- For each clause c of the form $\mathsf{p}(\vec{s}, \vec{t}) \,\mathsf{:-}\, \mathsf{p}_1(\vec{s_1}, \vec{t_1}), \ldots, \mathsf{p}_k(\vec{s_k}, \vec{t_k})$, the resulting TRS contains the following rules:

$$\begin{aligned}
\mathsf{p}^{\mathit{in}}(\vec{s}) &\to \mathsf{u}_1^c(\mathsf{p}_1^{\mathit{in}}(\vec{s}_1), \mathcal{V}(\vec{s})) \\
\mathsf{u}_1^c(\mathsf{p}_1^{\mathit{out}}(\vec{t}_1), \mathcal{V}(\vec{s})) &\to \mathsf{u}_2^c(\mathsf{p}_2^{\mathit{in}}(\vec{s}_2), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t}_1)) \\
&\ldots \\
\mathsf{u}_k^c(\mathsf{p}_k^{\mathit{out}}(\vec{t}_k), \mathcal{V}(\vec{s}) \cup \mathcal{V}(\vec{t}_1) \cup \ldots \cup \mathcal{V}(\vec{t}_{k-1})) &\to \mathsf{p}^{\mathit{out}}(\vec{t})
\end{aligned}$$

If the resulting TRS is terminating, then the original logic program terminates for any query with ground terms on all input positions of the predicates, cf. (Ohlebusch 2001). For our example program, we obtain the following TRS.

$$\begin{aligned}
\mathsf{app}^{\mathit{in}}(\mathit{Ys}) &\to \mathsf{app}^{\mathit{out}}([\,], \mathit{Ys}) \\
\mathsf{app}^{\mathit{in}}(.(X, \mathit{Zs})) &\to \mathsf{u}_1^{(2)}(\mathsf{app}^{\mathit{in}}(\mathit{Zs}), X, \mathit{Zs}) \\
\mathsf{u}_1^{(2)}(\mathsf{app}^{\mathit{out}}(\mathit{Xs}, \mathit{Ys}), X, \mathit{Zs}) &\to \mathsf{app}^{\mathit{out}}(.(X, \mathit{Xs}), \mathit{Ys}) \\
\mathsf{sublist}^{\mathit{in}}(Y) &\to \mathsf{u}_1^{(3)}(\mathsf{app}^{\mathit{in}}(Y), Y) \\
\mathsf{u}_1^{(3)}(\mathsf{app}^{\mathit{out}}(P, U), Y) &\to \mathsf{u}_2^{(3)}(\mathsf{app}^{\mathit{in}}(P), Y, P, U) \\
\mathsf{u}_2^{(3)}(\mathsf{app}^{\mathit{out}}(V, X), Y, P, U) &\to \mathsf{sublist}^{\mathit{out}}(X)
\end{aligned}$$

However, the complexity of this TRS is linear instead of quadratic. The reason is that backtracking in Prolog is replaced by non-deterministic choice in the TRS. While Prolog uses backtracking to traverse the whole SLD-tree, the evaluation of the TRS corresponds to exactly one branch in the tree. Since the SLD-tree is finitely branching, this is sound for termination analysis, but not for complexity. So we need a transformation which takes backtracking into account in order to make complexity analysis of TRSs applicable for complexity analysis of Prolog.

4 Constructing Derivation Graphs

We now explain the construction of *derivation graphs* which represent all evaluations of a Prolog program for a certain *class* of queries, cf. (Schneider-Kamp et al. 2010). Here, we regard *abstract* states, which represent sets of concrete states. In addition to the set of "ordinary" variables $\mathcal{N}$, we also use a set of abstract variables $\mathcal{A} = \{T_1, T_2, \ldots\}$ which represent fixed, but arbitrary terms (thus, $\mathcal{V} = \mathcal{N} \uplus \mathcal{A}$). To instantiate abstract variables, we use special substitutions γ (called *concretiza-*

[8] The same is true for the more refined transformation of (Schneider-Kamp et al. 2009) which works similarly, but which can also handle non-well-moded programs.

$$\frac{(t,Q) \mid S \;;\; \mathcal{G}}{(t,Q)^{c_1} \mid \cdots \mid (t,Q)^{c_a} \mid S \;;\; \mathcal{G}}\ (\textsc{Case}) \quad \text{if } \mathit{Slice}_{\mathcal{P}}(t) = (c_1,\ldots,c_a)$$

$$\frac{\Box \mid S \;;\; \mathcal{G}}{S \;;\; \mathcal{G}}\ (\textsc{Suc}) \qquad \frac{S \;;\; \mathcal{G}}{S' \;;\; \mathcal{G}'}\ (\textsc{Inst}) \quad \text{if there is a } \mu \text{ such that } S = S'\mu \text{ and } \mathcal{G} = \bigcup_{T\in\mathcal{G}'} \mathcal{V}(T\mu).$$

$$\frac{(t,Q)^{h \coloneqq B} \mid S \;;\; \mathcal{G}}{(B\sigma, Q\sigma) \mid S\sigma|_{\mathcal{G}} \;;\; \mathcal{G}' \qquad S \;;\; \mathcal{G}}\ (\textsc{Eval})$$

where $\mathit{mgu}(t,h) = \sigma$. W.l.o.g., for all $X \in \mathcal{V}$, $\mathcal{V}(\sigma(X))$ only contains fresh abstract variables not occurring in t, Q, S, or $\mathcal{G}$. Moreover, we have $\mathcal{G}' = \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}}))$.

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

where δ replaces all (abstract and non-abstract) variables from $\mathcal{V} \setminus \mathcal{G}$ by fresh abstract variables and $\mathcal{G}' = \mathcal{G} \cup \mathit{NextG}(t,\mathcal{G})\delta$, i.e., $\mathcal{G}$ is extended by the δ-renamings of those variables which will be instantiated by a ground term after each successful evaluation of t.

Fig. 2. Inference Rules for Abstract States

tions) where $\mathit{Dom}(\gamma) = \mathcal{A}$ and $\mathit{Range}(\gamma) \subseteq \mathcal{T}(\Sigma, \mathcal{N})$. Apart from the sequence of goals, an abstract state contains a set $\mathcal{G} \subseteq \mathcal{A}$ of abstract variables that only represent ground terms (in the derivation graph, we denote such variables by overlining them). So we only consider concretizations γ where $\gamma(T)$ is ground for all $T \in \mathcal{G}$.

In Fig. 2 we extend the inference rules of our operational semantics from Sect. 2 to abstract states. For the rules Suc and Case, this is straightforward. For Eval, however, note that an abstract state may represent both concrete states where the unification of the current query t with the head h of the next program clause succeeds or fails. Thus, the abstract Eval rule has two successor states in order to combine both the concrete Eval and the concrete Backtrack rule. Consequently, we obtain derivation trees instead of derivation sequences.

In Eval, we assume that $\mathit{mgu}(t,h) = \sigma$ renames all variables to fresh abstract variables (to handle sharing effects correctly). If a concretization γ corresponds to Eval's first successor (i.e., if $t\gamma$ and h unify), 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 the remaining goals S. The new set $\mathcal{G}'$ of abstract variables that may only be instantiated by ground terms are the abstract variables occurring in $\mathit{Range}(\sigma|_{\mathcal{G}})$. Fig. 3 shows the derivation for our example program when called with queries of the form $\mathsf{sublist}(T_1, \overline{T_2})$ (i.e., the initial state A corresponds to the class of queries $\mathcal{Q}_m^{\mathsf{sublist}}$ where sublist's second argument is ground). The nodes of such a derivation graph are states and each step from a node to its children is done by the inference rules of Fig. 2.

In Fig. 3, as the child of D, we have the state $(\langle \mathsf{app}(T_{11}, T_8, \overline{T_9})^{(1)} \mid \mathsf{app}(T_{11}, T_8, \overline{T_9})^{(2)} \rangle \;;\; \mathcal{G})$ where $\mathcal{G} = \{\overline{T_9}\}$. Here, `app([],Ys,Ys)` must be used for the next evaluation. The Eval rule yields two successors: In the first, we have $\sigma = \mathit{mgu}(\mathsf{app}(T_{11}, T_8, \overline{T_9}), \mathsf{app}([\,], Ys, Ys)) = \{T_8/\overline{T_{12}}, \overline{T_9}/\overline{T_{12}}, T_{11}/[\,], Ys/\overline{T_{12}}\}$ which leads to $(\langle \Box \mid \mathsf{app}(T_{11}, T_8, \overline{T_{12}})^{(2)} \rangle \;;\; \{\overline{T_{12}}\})$. The second successor is $(\langle \mathsf{app}(T_{11}, T_8, \overline{T_9})^{(2)} \rangle \;;\; \mathcal{G})$.

If one uses the Eval rule for a state s, then we say that the mgu σ is *associated* to the node s and label the edge to its first successor by σ. In these labels, we restrict the substitutions to those variables occurring in the state. So in Fig. 3, the substitution $\{T_8/\overline{T_{12}}, \overline{T_9}/\overline{T_{12}}, T_{11}/[\,]\}$ is associated to the child of node D.

To represent all possible evaluations in a finite way, we need additional inference rules to obtain finite derivation graphs instead of infinite derivation trees. To this end, we use an inference rule which can refer back to already existing states. Such

INST edges can be drawn in the derivation graph if the current state s represents a subset of those concrete states that are represented by an already existing state s' (i.e., s is an *instance* of s'). Essentially, this holds if there is a matching substitution μ making s' equal to s. Moreover, s and s' must have the same groundness information (modulo μ). Then we say that μ is *associated* to s and label the INST edge from s to s' by μ. So $\mu = \{T_{11}/T_{16}, T_8/T_{13}, \overline{T_9}/\overline{T_{15}}\}$ is associated to H and the edge from H to D is labeled with μ.

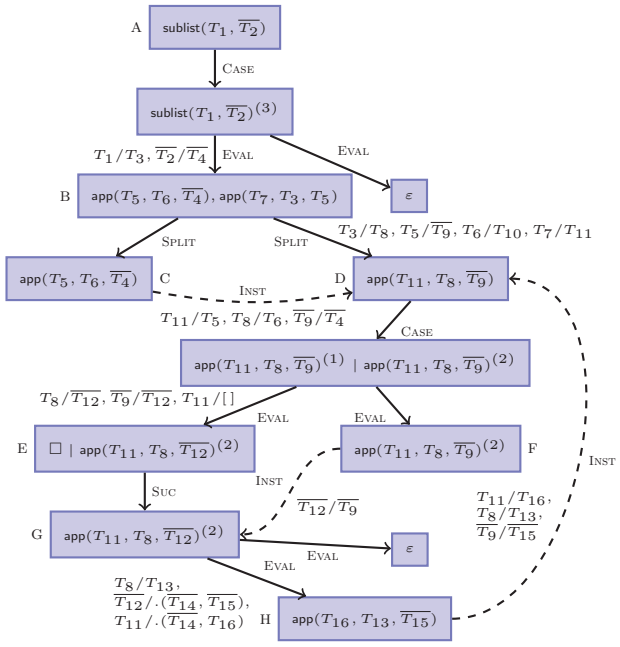

Fig. 3. Derivation Graph for the `sublist` Program

Moreover, we also need a SPLIT inference rule which splits up queries to make the INST rule applicable. In our example, we split the query (app$(T_5, T_6, \overline{T_4})$, app$(T_7, T_3, T_5)$) in state B. Otherwise, when evaluating the first atom app$(T_5, T_6, \overline{T_4})$ by the program clause (2), we use the substitution $\{T_5/.(\overline{T_{12}}, T_{14}), T_6/T_{15}, \overline{T_4}/.(\overline{T_{12}}, \overline{T_{13}}), T_7/T_{16}, T_3/T_{10}\}$ and reach a state with the query app$(T_{14}, T_{15}, \overline{T_{13}})$, app$(T_{16}, T_{10}, .(\overline{T_{12}}, T_{14}))$. But this new state is no instance of the state B, as we would need to match T_5 both to T_{14} and to $.(\overline{T_{12}}, T_{14})$. So without splitting queries, we would get an infinite derivation where no resulting state is an instance of a former state.

When splitting away the first atom t of a query, we over-approximate the possible answer substitutions for t by a substitution δ.[9] While δ is just a variable renaming of the abstract variables, we use *groundness analysis* (see e.g., (Howe and King 2003)) to infer a set $\mathit{NextG}(t, \mathcal{G})$ of abstract variables of t which are instantiated to ground terms in every successful derivation starting from a concretization of t. More precisely, let $\mathit{Ground}_{\mathcal{P}} : \Sigma \times 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ be a groundness analysis function. So if $\mathsf{p}/n \in \Sigma$, $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$, and $\mathit{Ground}_{\mathcal{P}}(\mathsf{p}, \{i_1, \ldots, i_m\}) = \{j_1, \ldots, j_k\}$, then any successful derivation of $\mathsf{p}(t_1, \ldots, t_n)$ where $t_{i_1}, \ldots, t_{i_m}$ are ground leads to an answer substitution θ where $t_{j_1}\theta, \ldots, t_{j_k}\theta$ are ground. Thus, $\mathit{Ground}_{\mathcal{P}}$ approximates which positions of p will become ground if the "input" positions $i_1, \ldots, i_m$ are ground. Then, we define $\mathit{NextG}(\mathsf{p}(t_1, \ldots, t_n), \mathcal{G}) = \{\mathcal{V}(t_j) \mid j \in \mathit{Ground}_{\mathcal{P}}(\mathsf{p}, \{i \mid \mathcal{V}(t_i) \subseteq \mathcal{G}\})\}$. In the SPLIT rule, the variables in $\mathit{NextG}(t, \mathcal{G})$ are renamed according to δ and added to the set $\mathcal{G}$ of abstract variables representing ground terms.

In our example, we infer that every successful evaluation of app$(T_5, T_6, \overline{T_4})$ instan-

[9] The SPLIT rule is only applicable to states containing just a single goal. In our implementation, we use an additional inference rule to split up sequences of goals, but we omitted it in the paper for readability. See (Schneider-Kamp et al. 2010) and the appendix for more details.

tiates the terms represented by T_5 and T_6 to ground terms. If δ is a renaming with $\delta = \{T_3/T_8, T_5/\overline{T_9}, T_6/\overline{T_{10}}, T_7/T_{11}\}$, we have $NextG(\mathsf{app}(T_5, T_6, \overline{T_4}), \mathcal{G})\delta = \{\delta(T_5), \delta(T_6)\} = \{\overline{T_9}, \overline{T_{10}}\}$. So while the first successor of the SPLIT rule has the query $\mathsf{app}(T_5, T_6, \overline{T_4})$, the second successor has the query $\mathsf{app}(T_{11}, T_8, \overline{T_9})$ where $\overline{T_9}$ only represents ground terms. We say that δ is *associated* to the node where we applied the SPLIT rule and we label the edge from this node to its second successor with δ. So in our example, δ is associated to B and the edge from B to D is labeled with δ.

See (Schneider-Kamp et al. 2010) for more details, further inference rules (in order to handle also non-definite programs), and more explanation on the graph construction. We always require that derivation graphs are finite, that they may not contain cycles consisting only of INST edges, and that all leaves of the graph are states with empty sequences ε of goals. Note that the derivation graph[10] in Fig. 3 is already an over-approximation of the original program since rules like EVAL or SPLIT may introduce abstract states representing concrete states which are not reachable from the initial class of queries.

To obtain a transformation which over-approximates the complexity of the original program (i.e., where the innermost runtime complexity of the resulting TRS is an upper bound for the complexity of the Prolog program), we encode the *paths* of the derivation graph. In this way, we can represent backtracking explicitly.

5 Complexity Analysis by Synthesizing TRSs from Derivation Graphs

In Sect. 5.1 we first present our approach to generate TRSs from derivation graphs. Afterwards, in Sect. 5.2 we show how to use these TRSs in order to obtain an upper bound on the complexity of the original Prolog program.

5.1 Synthesizing TRSs from Derivation Graphs

For a derivation graph G and an inference rule RULE, let $Rule(G)$ denote all nodes of G to which RULE has been applied. We denote by $Succ_i(s)$ the i-th child of node s and by $Succ_i(Rule(G))$ the set of i-th children of all nodes from $Rule(G)$.

To obtain a TRS from G, we encode the states as terms. For each state s, we use two fresh function symbols f_s^{in} and f_s^{out}. The arguments of f_s^{in} are the abstract variables in $\mathcal{G}$ (which represent ground terms). The arguments of f_s^{out} are those abstract variables which will be instantiated by ground terms after the successful evaluation of the query in s. To determine them, we again use groundness analysis. Formally, the encoding of states is done by two functions ren^{in} and ren^{out}. For B, we obtain $ren^{in}(\text{B}) = f_{\text{B}}^{in}(T_4)$ (since $\mathcal{G} = \{T_4\}$ in B) and $ren^{out}(\text{B}) = f_{\text{B}}^{out}(T_5, T_6, T_7, T_3)$ (since every successful evaluation of $(\mathsf{app}(T_5, T_6, T_4), \mathsf{app}(T_7, T_3, T_5))$ where T_4 is instantiated by a ground term instantiates T_5, T_6, T_7, T_3 by ground terms as well).

For an INST node (i.e., a node like C which has an INST edge labeled by a matching substitution μ to another node D), we do not introduce fresh function symbols. Instead, we take the terms resulting from its successor D, but we apply the match-

[10] The application of inference rules to abstract states is not deterministic and, thus, we may obtain a different derivation graph if we use a different heuristic for the application of the rules.

ing substitution μ to them. In other words, we have $ren^{in}(\text{C}) = ren^{in}(\text{D})\,\mu = f^{in}_{\text{D}}(T_9)\,\mu = f^{in}_{\text{D}}(T_4)$ and $ren^{out}(\text{C}) = ren^{out}(\text{D})\,\mu = f^{out}_{\text{D}}(T_{11}, T_8)\,\mu = f^{out}_{\text{D}}(T_5, T_6)$.

Definition 4 (*Encoding States as Terms*)
For an abstract state $s = (S; \mathcal{G})$, we define the functions ren^{in} and ren^{out} by:

$$ren^{in}(s) = \begin{cases} ren^{in}(Succ_1(s))\,\mu, & \text{if } s \in Inst(G) \text{ where } \mu \text{ is associated to } s \\ f^{in}_s(\mathcal{G}^{in}(s)), & \text{otherwise, where } \mathcal{G}^{in}(S;\mathcal{G}) = \mathcal{G} \cap \mathcal{V}(S) \end{cases}$$

$$ren^{out}(s) = \begin{cases} ren^{out}(Succ_1(s))\,\mu, & \text{if } s \in Inst(G) \text{ where } \mu \text{ is associated to } s \\ f^{out}_s(\mathcal{G}^{out}(s)), & \text{otherwise, where}^{11}\ \mathcal{G}^{out}((t_1,\ldots,t_k);\mathcal{G}) \\ & = NextG((t_1,\ldots,t_k), \mathcal{G} \cap \mathcal{V}(S)) \end{cases}$$

Here, we extended NextG to work not only on atoms, but also on queries:

$$NextG((t_1,\ldots,t_k),\mathcal{G}) = NextG(t_1,\mathcal{G}) \ \cup\ NextG(\,(t_2,\ldots,t_k),\, \mathcal{G} \cup NextG(t_1,\mathcal{G})\,).$$

So to compute $NextG((t_1,\ldots,t_k),\mathcal{G})$ for a query $(t_1,\ldots,t_k)$, in the beginning we only know that the abstract variables in $\mathcal{G}$ represent ground terms. Then we compute the variables $NextG(t_1,\mathcal{G})$ which are instantiated by ground terms after successful evaluation of t_1. Next, we compute the variables $NextG(t_2,\, \mathcal{G} \cup NextG(t_1,\mathcal{G}))$ which are instantiated by ground terms after successful evaluation of t_2, etc.

Now we encode the paths of G as rewrite rules. However, we only consider certain *connection paths* of G which suffice to approximate the complexity of the program. Connection paths are non-empty paths that start in the root node of the graph or in a successor state of an Inst or Split node, provided that these states are not Inst or Split nodes themselves. So the start states in our example are A, D, and G. Moreover, connection paths end in an Inst, Split, or Suc node or in the successor of an Inst node, while not traversing Inst or Split nodes or successors of Inst nodes in between. So in our example, the end states are B, C, D, E, F, G, H, but apart from E, the paths may not traverse any of these end nodes.

Thus, we have connection paths from A to B, from D to E, from D to F, from D to G, and from G to H. These paths cover all ways through the graph except for Inst edges (which are covered by the encoding of states to terms), for graph parts without cycles or Suc nodes (which are irrelevant since they represent evaluations which fail in constant time), and for Split edges (which we consider later in Def. 7).

Definition 5 (*Connection Path*)
A path $\pi = s_1 \ldots s_k$ is a connection path *of a derivation graph G iff $k > 1$ and*

- $s_1 \in \{root(G)\} \cup Succ_1(Inst(G) \cup Split(G)) \cup Succ_2(Split(G))$
- $s_k \in Inst(G) \cup Split(G) \cup Suc(G) \cup Succ_1(Inst(G))$
- *for all* $1 \le j < k$, $s_j \notin Inst(G) \cup Split(G)$
- *for all* $1 < j < k$, $s_j \notin Succ_1(Inst(G))$

This consideration of paths is similar to our approaches for termination analysis (Schneider-Kamp et al. 2010; Ströder et al. 2010), but now the paths are used to generate a TRS instead of a logic program. Moreover, for complexity analysis we

[11] To ease readability, in the definition of $\mathcal{G}^{out}$ we restricted ourselves to states with only one goal. See the appendix for a definition considering also states with sequences of goals.

need a more sophisticated treatment of SPLIT nodes than for termination analysis. The reason is that for termination, we only have to approximate the form of the answer substitutions that are computed for the first successor of a SPLIT node. This suffices to analyze termination of the evaluations starting in the second successor. For complexity analysis, however, we also need to know *how many* answer substitutions are computed for the first successor of a SPLIT node, since the evaluation of the second successor is repeated for each such answer substitution.

To convert connection paths to rewrite rules, the idea is to consider a path as a clause, where the first state of the path is the clause head, the last state of the path is the clause body, and we apply all substitutions along the path to the clause head. For instance, the connection path from A to B is considered as a clause $\mathsf{sublist}(T_3, \overline{T_4}) \mathbin{:-} \mathsf{app}(T_5, T_6, \overline{T_4}), \mathsf{app}(T_7, T_3, T_5)$, where the head of the clause results from applying the substitution $\{T_1/T_3, \overline{T_2}/\overline{T_4}\}$ to the query in state A.

Then we construct TRSs similar to the direct transformation from Sect. 3. So if π is the connection path from A to B and if σ_π are the substitutions on its edges, then the rewrite rules corresponding to π evaluate the instantiated input term $ren^{in}(\text{A})\,\sigma_\pi$ for the start node A to its output term $ren^{out}(\text{A})\,\sigma_\pi$ provided that the input term $ren^{in}(\text{B})$ for the end node can be evaluated to its output term $ren^{out}(\text{B})$. Thus, we obtain the rules $ren^{in}(\text{A})\,\sigma_\pi \to \mathsf{u}_{\text{A,B}}(\, ren^{in}(\text{B}),\, \mathcal{V}(ren^{in}(\text{A})\,\sigma_\pi)\,)$ and $\mathsf{u}_{\text{A,B}}(\, ren^{out}(\text{B}),\, \mathcal{V}(ren^{in}(\text{A})\,\sigma_\pi)\,) \to ren^{out}(\text{A})\,\sigma_\pi$. In our example, this yields

$$f^{in}_{\text{A}}(T_4) \to \mathsf{u}_{\text{A,B}}(f^{in}_{\text{B}}(T_4), T_4) \tag{4a}$$

$$\mathsf{u}_{\text{A,B}}(f^{out}_{\text{B}}(T_5, T_6, T_7, T_3), T_4) \to f^{out}_{\text{A}}(T_3) \tag{4b}$$

However, connection paths π' like the one from D to E where the end node is a SUC node, are considered like a clause $\mathsf{app}([\,], \overline{T_{12}}, \overline{T_{12}}) \mathbin{:-} \square$, i.e., like a fact. Thus, here the resulting rewrite rule directly evaluates the instantiated input term $ren^{in}(\text{D})\,\sigma_{\pi'}$ for the start node D to its output term $ren^{out}(\text{D})\,\sigma_{\pi'}$. Thus, we obtain

$$f^{in}_{\text{D}}(T_{12}) \to f^{out}_{\text{D}}([\,], T_{12}) \tag{5}$$

The rewrite rules for the connection path from D to G encode that the SUC node E contains another goal which is evaluated as well (when backtracking). So instead of backtracking, in the TRS we have a non-deterministic choice to decide whether to apply Rule (5) or the Rules (6*a*) and (6*b*) when evaluating a term built with f^{in}_{D}.

$$f^{in}_{\text{D}}(T_{12}) \to \mathsf{u}_{\text{D,G}}(f^{in}_{\text{G}}(T_{12}), T_{12}) \tag{6a}$$

$$\mathsf{u}_{\text{D,G}}(f^{out}_{\text{G}}(T_{11}, T_8), T_{12}) \to f^{out}_{\text{D}}(T_{11}, T_8) \tag{6b}$$

Definition 6 (*Rules for Connection Paths*)
For a connection path $\pi = s_1 \dots s_k$, the substitution σ_π is obtained by composing all substitutions on the edges of the path. So formally, we define σ_π as follows (where σ is the associated substitution of the node s_{k-1} and id is the identical substitution):

$$\sigma_{s_1 \dots s_k} = \begin{cases} id, & \text{if } k = 1 \\ \sigma_{s_1 \dots s_{k-1}}\sigma, & \text{if } s_{k-1} \in \mathit{Eval}(G), s_k = \mathit{Succ}_1(s_{k-1}) \\ \sigma_{s_1 \dots s_{k-1}}, & \text{otherwise} \end{cases}$$

Moreover, we define the rewrite rules corresponding to π as follows. If $s_k \in$

Suc(G), *then* *ConnectionRules*$(\pi) = \{ren^{in}(s_1)\,\sigma_\pi \to ren^{out}(s_1)\,\sigma_\pi\}$. *Otherwise,*

$$\begin{array}{rcl} \mathit{ConnectionRules}(\pi) & = & \{\ ren^{in}(s_1)\,\sigma_\pi \quad \to \quad \mathsf{u}_{s_1,s_k}(\,ren^{in}(s_k),\, \mathcal{V}(ren^{in}(s_1)\,\sigma_\pi)\,), \\ & & \mathsf{u}_{s_1,s_k}(\,ren^{out}(s_k),\, \mathcal{V}(ren^{in}(s_1)\,\sigma_\pi)\,) \quad \to \quad ren^{out}(s_1)\,\sigma_\pi\ \}, \end{array}$$

where u_{s_1,s_k} *is a fresh function symbol.*

So in addition to the rules (4a), (4b), (5), (6a), (6b) above, we obtain the rules (7a) and (7b) for the path from D to F, and (8a) and (8b) for the path from G to H.

$$\begin{array}{ll} (7a) & f^{in}_{\mathrm{D}}(T_9) \to \mathsf{u}_{\mathrm{D,F}}(f^{in}_{\mathrm{G}}(T_9), T_9) \\ (7b) & \mathsf{u}_{\mathrm{D,F}}(f^{out}_{\mathrm{G}}(T_{11}, T_8), T_9) \to f^{out}_{\mathrm{D}}(T_{11}, T_8) \\ (8a) & f^{in}_{\mathrm{G}}(.(T_{14}, T_{15})) \to \mathsf{u}_{\mathrm{G,H}}(f^{in}_{\mathrm{D}}(T_{15}), T_{14}, T_{15}) \\ (8b) & \mathsf{u}_{\mathrm{G,H}}(f^{out}_{\mathrm{D}}(T_{16}, T_{13}), T_{14}, T_{15}) \to f^{out}_{\mathrm{G}}(.(T_{14}, T_{16}), T_{13}) \end{array}$$

In addition to the rules for the connection paths, we also need rewrite rules to simulate the evaluation of Split nodes like B. Let δ be the substitution associated to B (i.e., δ is a variable renaming used to represent the answer substitution of B's first successor C). Then the Split node B succeeds (i.e., $ren^{in}(\mathrm{B})\,\delta$ can be evaluated to $ren^{out}(\mathrm{B})\,\delta$) if both successors C and D succeed (i.e., $ren^{in}(\mathrm{C})\,\delta$ can be evaluated to $ren^{out}(\mathrm{C})\,\delta$ and $ren^{in}(\mathrm{D})$ can be evaluated to $ren^{out}(\mathrm{D})$). So we obtain

$$\begin{array}{ll} (9a) & f^{in}_{\mathrm{B}}(T_4) \to \mathsf{u}_{\mathrm{B,C}}(f^{in}_{\mathrm{D}}(T_4), T_4) \\ (9b) & \mathsf{u}_{\mathrm{B,C}}(f^{out}_{\mathrm{D}}(T_9, T_{10}), T_4) \to \mathsf{u}_{\mathrm{C,D}}(f^{in}_{\mathrm{D}}(T_9), T_4, T_9, T_{10}) \\ (9c) & \mathsf{u}_{\mathrm{C,D}}(f^{out}_{\mathrm{D}}(T_{11}, T_8), T_4, T_9, T_{10}) \to f^{out}_{\mathrm{B}}(T_9, T_{10}, T_{11}, T_8) \end{array}$$

Definition 7 (*Rules for Split Nodes, Corresponding TRS of a Derivation Graph*) *Let* $s \in \mathit{Split}(G)$, $s_1 = \mathit{Succ}_1(s)$, *and* $s_2 = \mathit{Succ}_2(s)$. *Moreover, let* δ *be the substitution associated to* s. *Then* *SplitRules*$(s) =$

$$\begin{array}{l} \{\ ren^{in}(s)\,\delta \ \to\ \mathsf{u}_{s,s_1}(\,ren^{in}(s_1)\,\delta,\, \mathcal{V}(ren^{in}(s)\,\delta)\,), \\ \quad \mathsf{u}_{s,s_1}(\,ren^{out}(s_1)\,\delta,\, \mathcal{V}(ren^{in}(s)\,\delta)\,) \ \to\ \mathsf{u}_{s_1,s_2}(\,ren^{in}(s_2),\, \mathcal{V}(ren^{in}(s)\,\delta) \cup \mathcal{V}(ren^{out}(s_1)\delta)\,), \\ \quad \mathsf{u}_{s_1,s_2}(\,ren^{out}(s_2),\, \mathcal{V}(ren^{in}(s)\,\delta) \cup \mathcal{V}(ren^{out}(s_1)\,\delta)\,) \ \to\ ren^{out}(s)\,\delta\ \}. \end{array}$$

So the TRS $\mathcal{R}(G)$ corresponding *to* G *consists of* *ConnectionRules*(π) *for all connection paths* π *of* G *and of* *SplitRules*(s) *for all* Split *nodes* s *of* G.

In our example, $\mathcal{R}(G) = \{(4a), (4b), (5), (6a), (6b), (7a), (7b), (8a), (8b), (9a), (9b), (9c)\}$.

5.2 Using TRSs for Complexity Analysis of Prolog *Programs*

By the approach of Sect. 5.1, we can now automatically generate a TRS from a Prolog program. However, for complexity analysis, this TRS still has similar drawbacks as the one obtained by the direct transformation of Sect. 3. The problem is that the evaluation with the TRS still does not simulate the traversal of the whole SLD tree by backtracking. So the innermost runtime complexity for the TRS $\mathcal{R}$ with the rules (4a), (4b), ..., (9a), (9b), (9c) is only linear whereas the runtime complexity of the original Prolog program is quadratic.

The problem is due to the Split nodes of the derivation graph. If the first successor of a Split node (i.e., a node like C) has k answer substitutions, then the evaluation of the second successor of the Split node (i.e., the evaluation of D) is repeated k times. Currently, this is not reflected in the TRS.

To solve this problem, we now generate two *separate* TRSs $\mathcal{R}_{\text{C}}$ and $\mathcal{R}_{\text{D}}$ for the subgraphs starting in the two successors C and D of a SPLIT node like B, and multiply their corresponding complexity functions $irc_{\mathcal{R}_{\text{C}},\mathcal{R}}$ and $irc_{\mathcal{R}_{\text{D}},\mathcal{R}}$. Here, $irc_{\mathcal{R}_{\text{C}},\mathcal{R}}$ differs from the ordinary innermost runtime complexity function $irc_{\mathcal{R}}$ by only counting those rewrite steps that are done with the sub-TRS $\mathcal{R}_{\text{C}} \subseteq \mathcal{R}$.

So in general, for any $\mathcal{R}' \subseteq \mathcal{R}$, the function $irc_{\mathcal{R}',\mathcal{R}}$ maps any $n \in \mathbb{N}$ to the maximal number of $\xrightarrow{i}_{\mathcal{R}'}$-steps that occur in any sequence of $\xrightarrow{i}_{\mathcal{R}}$-steps starting with a basic term t where $|t| \leq n$. Related notions of "relative" complexity for TRSs were used in (Avanzini and Moser 2009; Hirokawa and Moser 2008; Noschinski et al. 2011; Zankl and Korp 2010), for example. Existing automated complexity provers like AProVE can also approximate $irc_{\mathcal{R}',\mathcal{R}}$ asymptotically.

The function $irc_{\mathcal{R}_{\text{C}},\mathcal{R}}$ indeed yields an upper bound for the number k of answer substitutions for C, because the number of answer substitutions cannot be larger than the number of evaluation steps. In our example, both the runtime and the number of answer substitutions for the call $\mathsf{app}(T_5, T_6, \overline{T_4})$ in node C is linear in the size of $\overline{T_4}$'s concretization. Thus, the call $\mathsf{app}(T_{11}, T_8, \overline{T_9})$ in node D, which has linear runtime itself, needs to be repeated a linear number of times. Thus, by multiplying the linear runtime complexities of $irc_{\mathcal{R}_{\text{C}},\mathcal{R}}$ and $irc_{\mathcal{R}_{\text{D}},\mathcal{R}}$, we obtain the correct result that the runtime of the original Prolog program is (at most) quadratic.

Note that if the first successor C of a SPLIT node only had a *constant* number k of answer substitutions (i.e., if k did not depend on the size of C's arguments), then instead of multiplying the runtimes of the two TRSs $\mathcal{R}_{\text{C}}$ and $\mathcal{R}_{\text{D}}$ for the successors of the SPLIT node, it would be sufficient to *add* them. Since such an addition is already encoded in the *SplitRules* of Def. 7, we do not need to consider separate TRSs for the successors of such SPLIT nodes. We call a SPLIT node *multiplicative* if the number of answer substitutions of its first successor is not bounded by a constant and let $mults(G)$ be the set of all *multiplicative* SPLIT nodes of G. So in our example, $mults(G) = \{\text{B}\}$. We will present a sufficient syntactic criterion to detect non-multiplicative SPLIT nodes in Def. 13.

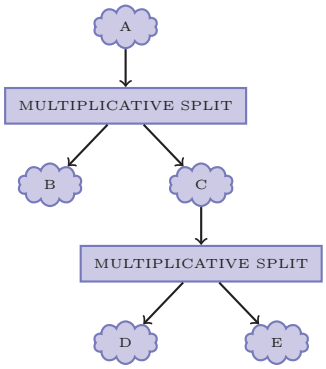

Fig. 8. Decomposing Graphs

So in order to infer an upper bound on the complexity of a Prolog program, we use the multiplicative SPLIT nodes of its derivation graph G to decompose G into subgraphs, such that multiplicative SPLIT nodes only occur as the leaves of subgraphs. For example, consider Fig. 8 where a derivation graph has been decomposed into the subgraphs $A, \ldots, E$ (the subgraphs A and C include the respective multiplicative SPLIT node as one of its leaves). We now determine the runtime complexities $irc_{\mathcal{R}(G_A),\mathcal{R}(G)}, \ldots, irc_{\mathcal{R}(G_E),\mathcal{R}(G)}$ separately and then we combine them in order to obtain an upper bound for the runtime of the whole Prolog program. As discussed above, the runtime complexity functions resulting from subgraphs of a multiplicative SPLIT node have to be multiplied. In contrast, the runtimes of subgraphs above a multiplicative SPLIT node have to be added. So for the graph in Fig. 8, we obtain $irc_{\mathcal{R}_A(G),\mathcal{R}(G)}(n) + irc_{\mathcal{R}_B(G),\mathcal{R}(G)}(n) \cdot$

$(irc_{\mathcal{R}_C(G),\mathcal{R}(G)}(n) + irc_{\mathcal{R}_D(G),\mathcal{R}(G)}(n) \cdot irc_{\mathcal{R}_E(G),\mathcal{R}(G)}(n))$ as an approximation for the complexity of the Prolog program.

To ensure that the derivation graph can indeed be decomposed into subgraphs as desired, we have to ensure that no multiplicative Split node can reach itself again.

Definition 9 (*Decomposable Derivation Graphs*)
A derivation graph G is called decomposable *iff there is no non-empty path from a node $s \in mults(G)$ to itself.*

The graph in Fig. 3 is indeed decomposable. However, decomposability is a real restriction and there are programs in the TPDB whose complexity we cannot analyze, because our graph construction yields a non-decomposable derivation graph.

Now for any node s, the *subgraph at node s* is the subgraph which starts in s and stops when reaching multiplicative Split nodes.

Definition 10 (*Subgraphs of Derivation Graphs*)
Let G be a decomposable derivation graph with nodes V and edges E (i.e., $G = (V, E)$) and let $s \in V$. Then we define the subgraph of G at node s *as the minimal graph $G_s = (V_s, E_s)$ where $s \in V_s$ and whenever $s_1 \in V_s \setminus mults(G)$ and $(s_1, s_2) \in E$, then $s_2 \in V_s$ and $(s_1, s_2) \in E_s$.*

Now we decompose the derivation graph into the subgraph at the root node and into the subgraphs at all successors of multiplicative Split nodes. So the graph in Fig. 3 is decomposed into G_{A}, G_{C}, and G_{D}, where G_{A} contains the 4 nodes from A to B and to ε, G_{C} contains all other nodes, and G_{D} contains all nodes of G_{C} except C.

Here, $\mathcal{R}(G_{\text{A}})$ consists of $ConnectionRules(\pi)$ for the connection path π from A to B and of $SplitRules(\text{B})$, i.e., $\mathcal{R}(G_{\text{A}}) = \{(4a), (4b), (9a), (9b), (9c)\}$. For both subgraphs G_{C} and G_{D}, we get the same TRS, because C is an instance of D, i.e., $\mathcal{R}(G_{\text{C}}) = \mathcal{R}(G_{\text{D}}) = \{(5), (6a), (6b), (7a), (7b), (8a), (8b)\}$.

To obtain an upper bound for the complexity of the original logic program, we now combine the complexities of the sub-TRSs as discussed before. So we multiply the complexities resulting from subgraphs of multiplicative Split nodes, and add all other complexities. The function $cplx_s(n)$ approximates the runtime of the logic program which is represented by the subgraph of G at node s.

Definition 11 (*Complexity for Subgraphs*)
Let $G = (V, E)$ be a decomposable derivation graph. For any $s \in V$ and $n \in \mathbb{N}$, let

$$cplx_s(n) = \begin{cases} cplx_{Succ_1(s)}(n) \cdot cplx_{Succ_2(s)}(n), & \text{if } s \in mults(G) \\ irc_{\mathcal{R}(G_s),\mathcal{R}(G)}(n) + \Sigma_{s' \in mults(G) \cap G_s}\ cplx_{s'}(n), & \text{otherwise} \end{cases}$$

So in our example, we obtain:

$$\begin{aligned} cplx_{\text{A}}(n) &= irc_{\mathcal{R}(G_{\text{A}}),\mathcal{R}(G)}(n) + cplx_{\text{B}}(n) \\ &= irc_{\mathcal{R}(G_{\text{A}}),\mathcal{R}(G)}(n) + cplx_{\text{C}}(n) \cdot cplx_{\text{D}}(n) \\ &= irc_{\mathcal{R}(G_{\text{A}}),\mathcal{R}(G)}(n) + irc_{\mathcal{R}(G_{\text{C}}),\mathcal{R}(G)}(n) \cdot irc_{\mathcal{R}(G_{\text{D}}),\mathcal{R}(G)}(n) \end{aligned}$$

Thm. C 13 states that combining the complexities of the TRSs as in Def. 11 indeed yields an upper bound for the complexity of the original Prolog program.[12]

[12] All proofs can be found in the appendix.

Theorem 12 (*Complexity Analysis for* Prolog *Programs*)
Let $\mathcal{P}$ be a Prolog *program,* $\mathsf{p} \in \Sigma$*, m a moding function, and G a decomposable derivation graph for $\mathcal{P}$ and the queries $\mathcal{Q}^{\mathsf{p}}_m$. Then $prc_{\mathcal{P},\mathcal{Q}^{\mathsf{p}}_m}(n) \in \mathcal{O}(cplx_{root(G)}(n))$.*

For our example program, automated tools for complexity analysis of TRSs like AProVE automatically prove that[13] $irc_{\mathcal{R}(G_{\text{A}}),\mathcal{R}(G)}(n) \in \mathcal{O}(n)$, $irc_{\mathcal{R}(G_{\text{C}}),\mathcal{R}(G)}(n) \in \mathcal{O}(n)$, and $irc_{\mathcal{R}(G_{\text{D}}),\mathcal{R}(G)}(n) \in \mathcal{O}(n)$. This implies $cplx_{\text{A}}(n) = irc_{\mathcal{R}(G_{\text{A}}),\mathcal{R}(G)}(n) + irc_{\mathcal{R}(G_{\text{C}}),\mathcal{R}(G)}(n) \cdot irc_{\mathcal{R}(G_{\text{D}}),\mathcal{R}(G)}(n) \in \mathcal{O}(n^2)$ and, thus, also $prc_{\mathcal{P},\mathcal{Q}^{\mathsf{sublist}}_m}(n) \in \mathcal{O}(n^2)$.

It remains to explain how to automatically identify non-multiplicative Split nodes. To this end, we have to prove that the number of answer substitutions for the first successor of a Split node is bounded by a constant. In our implementation, we use a sufficient criterion which can easily be checked automatically, cf. Def. 13 and Thm. 14. As future work, we could improve our analysis by combining it with other tools for determinacy analysis (e.g., (Kriener and King 2011; López-García et al. 2005; Mogensen 1996; Sahlin 1991)). These tools can prove upper bounds on the number of answer substitutions for a given class of queries.

Definition 13 (*Determinacy Criterion*)
A node s in G satisfies the determinacy criterion *if condition (a) or (b) holds:*

(a) *All successors of s satisfy the determinacy criterion. Moreover, if $s \in Suc(G)$, then there is no non-empty path from s to a* Suc *node in G.*
(b) *The node s is a* Split *node and at least one of $Succ_1(s)$ or $Succ_2(s)$ cannot reach a* Suc *node in G.*

The following theorem shows that the above determinacy criterion can indeed be used to detect Split nodes that are not multiplicative.

Theorem 14 (*Soundness of Determinacy Criterion*)
Let G be a complexity graph. Let s be a node in G which satisfies the determinacy criterion of Def. 13. Then for any concretization of s, its evaluation results in at most one answer substitution. Thus if s' is a Split *node and $Succ_1(s')$ satisfies the determinacy criterion, then s' is not multiplicative.*

6 Experiments and Conclusion

We proposed a new method to determine asymptotic upper bounds for the runtime complexity of Prolog programs automatically, based on a transformation to term rewriting. First, we showed that the existing transformations from logic programs to TRSs can yield a TRS whose runtime complexity is not an asymptotic upper bound for the runtime complexity of the original logic program. Thus, we presented a novel transformation where each asymptotic upper bound for the runtime complexity of the resulting TRS is also an upper bound for the runtime complexity of the original logic program. This transformation is also applicable to non-well-moded logic programs and programs using built-in predicates like cuts. For this transformation, we also developed a new criterion for determinacy of Prolog programs, based on deriva-

[13] Note that we even have $irc_{\mathcal{R}(G_{\text{A}}),\mathcal{R}(G)}(n) \in \mathcal{O}(1)$, i.e., the linear bound found by AProVE is not tight. This indicates that our approach does not always yield precise bounds. However, most bounds detected in our experiments are in fact tight.

	AProVE	CASLOG	CiaoPP `steps_ub`	CiaoPP `res_steps`
$\mathcal{O}(1)$	**54**	1	3	3
$\mathcal{O}(n)$	**108**	21	19	18
$\mathcal{O}(n^2)$	**42**	4	4	4
$\mathcal{O}(n \cdot 2^n)$	0	**3**	**3**	**3**
Total bounds	**204**	29	29	28
Runtime in s	6122	7042	**5579**	5953

Table 1. *Results on all 477 programs from the Termination Problem Data Base*

tion graphs. We implemented the transformation in our fully automated termination and complexity prover AProVE (Giesl et al. 2006). To compare its power and performance to existing direct approaches for cost analysis of Prolog, we evaluated it against the Complexity Analysis System for LOGic (CASLOG) (Debray and Lin 1993) and against the Ciao Preprocessor (CiaoPP) (Bueno et al. 2004), which implements the approach of (López-García et al. 2010). To this end, we ran the three tools on all 477 Prolog programs from the Termination Problem Data Base. For CiaoPP we used both the original cost analysis ("`steps_ub`") and CiaoPP's new resource framework which allows to measure different forms of costs. Here, we chose the cost measure "`res_steps`" which approximates the number of resolution steps needed in evaluations. Moreover, we also used CiaoPP to infer the mode and measure information required by CASLOG. The experiments were run on 2.2 GHz Quad-Opteron 848 Linux machines with a timeout of 60 seconds per program (as in the competition on automated complexity analysis).

Table 1 shows the results of our experiments with one column for each tool. The first four rows give the number of programs that could be shown to have a constant bound ($\mathcal{O}(1)$), a linear or quadratic polynomial bound ($\mathcal{O}(n)$ or $\mathcal{O}(n^2)$), or an exponential bound ($\mathcal{O}(n \cdot 2^n)$). In Rows 5 and 6 we give the total number of upper bounds that could be found by the tool and its total runtime on the whole example set, respectively. We highlight the best tool for each row using bold font. For the details of this empirical evaluation and to run AProVE via a web interface, we refer to `http://aprove.informatik.rwth-aachen.de/eval/plcost/`.This website also contains an extended version of the paper with all proofs the appendix.

The table shows that AProVE can find upper bounds for a much larger subset ($> 42\%$) of the programs than any of the other tools ($\approx 6\%$). However, there are also 9 examples where CASLOG or CiaoPP can prove constant (1), linear (5), or exponential bounds (3), whereas AProVE fails (5) or finds a weaker bound (4). In summary, the experiments clearly demonstrate that our transformational approach for determining upper bounds advances the state of the art in automated complexity analysis of logic programs significantly.

Acknowledgements. We thank M. Hermenegildo and P. López-García for their dedicated support. Without it, the experimental comparison with CASLOG and CiaoPP would not have been possible. We also thank N.-W. Lin for agreeing to make the updated version of CASLOG (running under Sicstus 4 or Ciao) available on our paper's web page.

References

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

Avanzini, M., Moser, G., and Schnabl, A. 2008. Automated implicit computational complexity analysis. In *Proc. IJCAR '08.* LNAI 5195. 132–138.

Avanzini, M. and Moser, G. 2009. Dependency pairs and polynomial path orders. In *Proc. RTA '09.* LNCS 5595. 48–62.

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That.* Cambridge University Press.

Bonfante, G., Cichon, A., Marion, J.-Y., and Touzet, H. 2001. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming 11,* 1, 33–53.

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., and Puebla, G. 2004. The Ciao system. Tech. rep., UPM. Available from `http://www.ciaohome.org`.

Debray, S. K. and Lin, N.-W. 1993. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems 15*, 826–875.

Debray, S. K., López-García, P., Hermenegildo, M. V., and Lin, N.-W. 1997. Lower bound cost estimation for logic programs. In *Proc. ILPS '97.* MIT Press, 291–305.

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

Hill, P. M. and King, A. 1997. Determinacy and determinacy analysis. *Journal of Programming Languages 5,* 1, 135–171.

Hirokawa, N. and Moser, G. 2008. Automated complexity analysis based on the dependency pair method. In *Proc. IJCAR '08.* LNAI 5195. 364–379.

Howe, J. M. and King, A. 2003. Efficient groundness analysis in Prolog. *Theory and Practice of Logic Programming 3,* 1, 95–124.

ISO/IEC 13211-1. 1995. *Information technology - Programming languages - Prolog.*

King, A., Shen, K., and Benoy, F. 1997. Lower-bound time-complexity analysis of logic programs. In *Proc. ILPS '97.* MIT Press, 261–285.

Kriener, J. and King, A. 2011. RedAlert: Determinacy inference for Prolog. *Theory and Practice of Logic Programming 11,* 4-5, 537–553.

López-García, P., Bueno, F., and Hermenegildo, M. 2005. Determinacy analysis for logic programs using mode and type information. In *Proc. LOPSTR '05.* LNCS 3573. 19–35.

López-García, P., Darmawan, L., and Bueno, F. 2010. A framework for verification and debugging of resource usage properties. In *Technical Communications of ICLP '10.* LIPIcs 7. Dagstuhl Publishing, 104–113.

Marion, J.-Y. and Péchoux, R. 2008. Characterizations of polynomial complexity classes with a better intensionality. In *Proc. PPDP '08.* ACM Press, 79–88.

Mogensen, T. 1996. A semantics-based determinacy analysis for Prolog with cut. In *Proc. Ershov Memorial Conference '96.* LNCS 1181. 374–385.

Noschinski, L., Emmes, F., and Giesl, J. 2011. The dependency pair framework for automated complexity analysis of term rewrite systems. In *Proc. CADE '11.* LNAI 6803. 422–438.

Ohlebusch, E. 2001. Termination of logic programs: Transformational methods revisited. *Applicable Algebra in Engineering, Communication and Computing 12,* 1–2, 73–116.

Sahlin, D. 1991. Determinacy analysis for full Prolog. In *Proc. PEPM '91.* ACM Press, 23–30.

Schneider-Kamp, P. 2008. Static termination analysis for Prolog using term rewriting and SAT solving. Ph.D. thesis, RWTH Aachen.

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

Schneider-Kamp, P., Giesl, J., Ströder, T., Serebrenik, A., and Thiemann, R. 2010. Automated termination analysis for logic programs with cut. In *Proc. ICLP '10, Theory and Practice of Logic Programming 10,* 4-6, 365–381.

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

Ströder, T., Schneider-Kamp, P., and Giesl, J. 2010. Dependency triples for improving termination analysis of logic programs with cut. In *Proc. LOPSTR '10.* LNCS 6564. 184–199.

Ströder, T., Emmes, F., Schneider-Kamp, P., Giesl, J., and Fuhs, C. 2011. A linear operational semantics for termination and complexity analysis of ISO Prolog. In *Proc. LOPSTR '11.* To appear. Available from `http://aprove.informatik.rwth-aachen.de/eval/plcost/`.

Waldmann, J. 2010. Polynomially bounded matrix interpretations. In *Proc. RTA '10.* LIPIcs 6. Dagstuhl Publishing, 357–372.

Zankl, H. and Korp, M. 2010. Modular complexity analysis via relative complexity. In *Proc. RTA '10.* LIPIcs 6. Dagstuhl Publishing, 385–400.

In Appendix A, we first recapitulate the set of inference rules and the construction of derivation graphs for Prolog using built-in predicates.[14] Then we extend the definitions from the paper to consider this more general setting in Appendix B. Afterwards, we prove the correctness of our transformation in Appendix C and the correctness of our determinacy criterion in Appendix D.

Appendix A Full Graph Construction

This section recapitulates all existing work on derivation graphs as used in this paper. The contents of this section are taken from (Schneider-Kamp 2008; Schneider-Kamp et al. 2010; Ströder 2010; Ströder et al. 2010; Ströder et al. 2011).

The ISO standard for Prolog (ISO/IEC 13211-1 1995) defines a list of built-in predicates. According to this standard we define the set *BuiltInPredicates* as the set containing exactly the following symbols:

- abolish/1
- arg/3
- =:=/2
- =\=/2
- >/2
- >=/2
- </2
- =</2
- asserta/1
- assertz/1
- at_end_of_stream/0
- at_end_of_stream/1
- atom/1
- atom_chars/2
- atom_codes/2
- atom_concat/3
- atom_length/2
- atomic/1
- bagof/3
- call/1
- catch/3
- char_code/2
- char_conversion/2
- clause/2
- close/1
- close/2
- compound/1
- ,/2
- copy_term/2
- current_char_conversion/2
- current_input/1
- current_op/3
- current_output/1
- current_predicate/1
- current_prolog_flag/2
- !/0
- ;/2
- fail/0
- findall/3
- float/1
- flush_output/0
- flush_output/1
- functor/3
- get_byte/1
- get_byte/2
- get_char/1
- get_char/2
- get_code/1
- get_code/2
- halt/0
- halt/1
- ->/2
- integer/1
- is/2
- nl/0
- nl/1
- nonvar/1
- \+/1
- number/1
- number_chars/2
- number_codes/2
- once/1
- op/3
- open/3
- open/4
- peek_byte/1
- peek_byte/2
- peek_char/1
- peek_char/2
- peek_code/1
- peek_code/2
- put_byte/1
- put_byte/2
- put_char/1
- put_char/2
- put_code/1
- put_code/2
- read/1

[14] While the operational semantics in (Ströder et al. 2011) covers all built-in predicates from (ISO/IEC 13211-1 1995), only 26 of these built-in predicates are currently supported by the inference rules on abstract states.

- read/2
- read_term/2
- read_term/3
- repeat/0
- retract/1
- set_input/1
- set_output/1
- set_prolog_flag/2
- set_stream_position/2
- setof/3
- stream_property/2
- sub_atom/5
- @>/2
- @>=/2
- ==/2
- @</2
- @=</2
- \==/2
- throw/1
- true/0
- \=/2
- =/2
- unify_with_occurs_check/2
- =../2
- var/1
- write/1
- write/2
- write_canonical/1
- write_canonical/2
- write_term/2
- write_term/3
- writeq/1
- writeq/2

According to the execution model of Prolog as defined in (ISO/IEC 13211-1 1995), there are some special positions inside the terms of clause bodies or goals which may be executed. When referring to such a term, these positions are exactly those reachable from the root of the term by a path having only function symbols from the set $\mathit{GoalJunctors} = \{,/2,\ ;/2,\ ->/2\}$ except for the position itself. For the clause body or goal, these positions are all such positions in the terms belonging to the clause body or goal, respectively.

Definition A 1 (*Predication Position, Predication*)
Given a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ and a position $\mathit{pos} \in \mathit{Occ}(t)$, we call pos a *predication position* w.r.t. t iff for all positions $\mathit{pos}' \in \mathit{Occ}(t)$ with $\mathit{pos}' \lhd \mathit{pos}$ we have $\mathit{root}(t|_{\mathit{pos}'}) \in \mathit{GoalJunctors}$. Furthermore, we call $t|_{\mathit{pos}}$ a *predication* w.r.t. t. For a finite list L of terms $t_1, \ldots, t_k$ we also call every predication position $\mathit{pos}_i \in \mathit{Occ}(t_i)$ w.r.t. t_i a predication position w.r.t. L and $t_i|_{\mathit{pos}_i}$ a predication w.r.t. L.

Although we do not distinguish between predicate and function symbols, we do make a distinction between individual cuts to make their scope explicit. However, this distinction is only necessary and correct if the cuts in question are predications w.r.t. the goal to execute. Concerning comparison or unification of terms, we must not make such a distinction. So we define a set of labeled cut operators $\mathsf{Cuts} = \bigcup_{m \in \mathbb{N}} \{!_m/0\}$ which we will use in the following definitions of goals and their transformation used in the ISO standard. Thus, we have to deal with terms not only containing function symbols from a signature Σ, but also from Cuts. However, the latter may only occur in predication positions. For this reason we define special sets of terms we use in Prolog.

Definition A 2 (*Terms in Prolog*)
The terms we consider in Prolog are from the set $\mathit{PrologTerms}(\Sigma, \mathcal{V}) = \{t \in \mathcal{T}(\Sigma \cup \mathsf{Cuts}, \mathcal{V}) \mid \forall \mathit{pos} \in \mathit{Occ}(t) : t|_{\mathit{pos}} \in \mathsf{Cuts} \implies \mathit{pos} \text{ is a predication position}\}$. The definition of predication positions and predications is therefore extended to work also on terms from $\mathit{PrologTerms}(\Sigma, \mathcal{V})$. Analogously, we define the set of ground terms for Prolog as $\mathit{GroundTerms}(\Sigma) = \{t \in \mathcal{T}(\Sigma \cup \mathsf{Cuts}, \varnothing) \mid \forall \mathit{pos} \in \mathit{Occ}(t) : t|_{\mathit{pos}} \in \mathsf{Cuts} \implies \mathit{pos} \text{ is a predication position}\}$.

A *query* or *goal* is a sequence of terms, and the set of all queries is denoted by

$Goal(\Sigma, \mathcal{V})$. The empty query is denoted by $\Box$, and a non-empty query consisting of a Prolog-term t and a remaining query Q is denoted by (t, Q).

The set of all substitutions over Σ and $\mathcal{V}$ is denoted $Subst(\Sigma, \mathcal{V})$. If two terms t_1 and t_2 unify, but we are not interested in a specific unifier, we often write $t_1 \sim t_2$. Likewise, we sometimes write $t_1 \nsim t_2$ as a synonym for $mgu(t_1, t_2) = \mathit{fail}$.

In many situations we will consider substitutions which are equal on a certain set of variables, while they do not replace any other variables. We call such substitutions *restricted* to a certain set. The restriction of σ to a set of variables $\mathcal{V}' \subseteq \mathcal{V}$ (denoted $\sigma|_{\mathcal{V}'}$) is therefore defined as $\sigma|_{\mathcal{V}'}(X) = \sigma(X)$, if $X \in \mathcal{V}'$, and $\sigma|_{\mathcal{V}'}(X) = X$, otherwise. Finally, we often need variables which do not occur anywhere else. We call such variables *fresh variables* and denote by $\mathcal{V}_{fresh} \subseteq \mathcal{V}$ the subset of fresh variables. Analogously, we denote the subset of fresh abstract and non-abstract variables by $\mathcal{A}_{fresh}$ and $\mathcal{N}_{fresh}$, respectively.

Abstract variables represent arbitrary terms in general, but to describe classes of queries typically specified by a function symbol and argument positions which should be instantiated by ground terms, we need to constrain the terms by which the abstract variables may be instantiated. Additionally, we want to keep track of non-abstract variables which do not occur in the terms represented by abstract variables. Finally, due to failing unifications during the evaluation, we gather knowledge about non-unifiable terms. Therefore, we add a *knowledge base* representable by a triple $KB = (\mathcal{G}, \mathcal{F}, \mathcal{U})$ to a list of goals containing abstract terms and without candidates for answer substitutions where $\mathcal{G} \subseteq \mathcal{A}$, $\mathcal{F} \subseteq \mathcal{N}$, and $\mathcal{U} \subseteq \mathit{PrologTerms}(\Sigma, \mathcal{V}) \times \mathit{PrologTerms}(\Sigma, \mathcal{V})$. Here, $\mathcal{G}$ is the set of all abstract variables whose instantiations are restricted to ground terms, while $\mathcal{F}$ contains those non-abstract variables which may not occur in the terms represented by abstract variables. Moreover, $\mathcal{U}$ represents a set of pairs of terms, where a pair of terms (s, t) represents that s and t are not unifiable after instantiating the abstract variables, i.e., that we have $mgu(s\gamma, t\gamma) = \mathit{fail}$ for a given instantiation γ of the abstract variables. The set of *abstract states* $\mathit{AState}(\Sigma, \mathcal{N}, \mathcal{A})$ is a set of pairs $(S; KB)$ of a list of goals S without candidates for answer substitutions (i.e., the candidates for answer substitutions are dropped from the concrete goals) and a knowledge base KB.

To define which concrete states are represented by an abstract state, we introduce the notion of a *concretization*. A concretization is a substitution γ replacing all and only abstract variables in an abstract state while respecting the knowledge base $(\mathcal{G}, \mathcal{F}, \mathcal{U})$.[15] So for an abstract state $S; KB$ with $S = G_1 \mid \cdots \mid G_n$, we have $S\gamma = G_1\gamma \mid \cdots \mid G_n\gamma$. For a goal G with $G = t_1, \ldots, t_k$, we have $G\gamma = (t_1\gamma, \ldots, t_k\gamma)$ For a goal G with $G = (t_1, \ldots, t_k)^c$, we have $G\gamma = (t_1\gamma, \ldots, t_k\gamma)^c$, and for a goal $G = ?_m$, we have $G\gamma = G$. Moreover, we have $\gamma|_{\mathcal{A}} = \gamma$ and $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$. Also,

[15] Still, we omit the explicit representation of answer substitutions and candidates for them from the full operational semantics in (Ströder et al. 2011) since they do not contribute to the complexity.

abstract variables from $\mathcal{G}$ are only replaced by ground terms, i.e., $\mathit{Range}(\gamma|_{\mathcal{G}}) \subseteq \mathit{GroundTerms}(\Sigma)$. Likewise, γ may not introduce variables from $\mathcal{F}$. This can be specified by $\mathcal{F}(\mathit{Range}(\gamma)) = \varnothing$. Finally, for all pairs $(t, t') \in \mathcal{U}$ we need to ensure that $t\gamma$ and $t'\gamma$ do not unify, i.e., that $\mathit{mgu}(t\gamma, t'\gamma) = \mathit{fail}$. For an abstract state $(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$, we define the set of *concretized states* $\mathcal{CON}(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ as the set $\{S\gamma \mid \gamma \text{ is a concretization w.r.t. the knowledge base } (\mathcal{G}, \mathcal{F}, \mathcal{U})\}$.

The ISO standard (ISO/IEC 13211-1 1995) describes a preprocessing transformation for queries and goals which we mimic by the following function.

Definition A 3 (*Transformation of Goals*)
The function $\mathit{Transformed} : \mathit{Goal}(\Sigma, \mathcal{V}) \times \mathbb{N} \to \mathit{Goal}(\Sigma, \mathcal{V})$ is recursively defined by

$$
\begin{array}{lcl}
\mathit{Transformed}(\square, m) & = & \square \\
\mathit{Transformed}((x, L), m) & = & \mathsf{call}(x), \mathit{Transformed}(L, m) \text{ for } x \in \mathcal{V} \\
\mathit{Transformed}((!, L), m) & = & !_m, \mathit{Transformed}(L, m) \\
\mathit{Transformed}((!_{m'}, L), m) & = & !_m, \mathit{Transformed}(L, m) \\
\mathit{Transformed}((f(t_1, t_2), L), m) & = & f(\mathit{Transformed}(t_1, m), \mathit{Transformed}(t_2, m)), \\
 & & \mathit{Transformed}(L, m) \text{ for } f \in \mathit{GoalJunctors} \\
\mathit{Transformed}((s, L), m) & = & s, \mathit{Transformed}(L, m) \text{ for} \\
 & & s \in \mathit{PrologTerms}(\Sigma, \mathcal{V}) \setminus \mathcal{V} \text{ with} \\
 & & \mathit{root}(s) \notin \mathit{GoalJunctors} \cup \{!/0\} \cup \mathsf{Cuts}
\end{array}
$$

Sometimes we have to replace some abstract variables by fresh ones to handle sharing effects correctly. To this end, we use the following substitution.

Definition A 4 (*Replacement by Fresh Abstract Variables*)
We define $\alpha_{\mathcal{M}}$ for a set of variables $\mathcal{M}$ as follows:

$$
\alpha_{\mathcal{M}}(x) = \begin{cases} a & \text{if } x \in \mathcal{M} \setminus \mathcal{V}_{\mathit{fresh}} \text{ for } a \in \mathcal{A}_{\mathit{fresh}} \\ x & \mathit{otherwise} \end{cases}
$$

Now we state all abstract inference rules which have been developed in (Schneider-Kamp 2008; Schneider-Kamp et al. 2010; Ströder 2010; Ströder et al. 2010).

Definition A 5 (*Abstract Inference Rules*)

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

$$\frac{\mathsf{call}(x), Q \mid S; KB}{\varepsilon; KB}\ (\textsc{VariableError})$$

$$\frac{t, Q \mid S; KB}{\varepsilon; KB}\ (\textsc{UndefinedError}) \quad \text{where } \mathit{Slice}_{\mathcal{P}}(t) = \varnothing$$

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

$$\frac{t, Q \mid S; KB}{(t, Q)^{i_1}_m \mid \ldots \mid (t, Q)^{i_k}_m \mid ?_m \mid S; KB}\ (\textsc{Case}) \quad \text{where } m \in \mathbb{N} \text{ is fresh, } i_1 < \ldots < i_k, \text{ and } \mathit{Slice}_{\mathcal{P}}(t) = \{c_{i_1}, \ldots, c_{i_k}\} \neq \varnothing$$

$$\frac{\mathsf{call}(t'), Q \mid S; KB}{t'', Q \mid ?_m \mid S; KB}\ (\textsc{Call})$$

where $m \in \mathbb{N}$ is fresh, $t' \in \mathit{PrologTerms}(\Sigma, \mathcal{V}) \setminus \mathcal{V}$, t' has only finitely many predication positions, $\forall \mathit{pos} \in \mathit{Occ}(t') : \mathit{pos}$ is a predication position $\implies t'|_{\mathit{pos}} \notin \mathcal{A}$ and $t'' = \mathit{Transformed}(t', m)$

$$\frac{(t, Q)^i_m \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}\ (\textsc{Backtrack})$$

where $i \neq b$, $c_i = H_i \mathbin{:-} B_i$ and either $\mathit{mgu}(t, H_i) = \mathit{fail}$ or $\sigma = \mathit{mgu}(t, H_i)$ with $\exists a \in \mathcal{G} : a\sigma \notin \mathit{PrologTerms}(\Sigma, \mathcal{V})$ or $\mathcal{V}(\mathit{Range}(\sigma)) \subseteq \mathcal{V}_{\mathit{fresh}}$, $\mathcal{V}(\mathit{Range}(\sigma|_{\mathcal{G}})) \subseteq \mathcal{A}$ and $\exists (s, s') \in \mathcal{U} : \sigma' = \mathit{mgu}(s\sigma|_{\mathcal{G}}, s'\sigma|_{\mathcal{G}}) \wedge \mathit{Dom}(\sigma') \subseteq \mathcal{N} \wedge \sigma'\sigma' = \sigma'$

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

where $i \neq b$, $c_i = H_i \mathbin{:-} B_i$, $\mathit{mgu}(t, H_i) = \sigma$ with $\mathcal{V}(\mathit{Range}(\sigma)) \subseteq \mathcal{V}_{\mathit{fresh}}$, $\mathcal{V}(\mathit{Range}(\sigma|_{\mathcal{A}})) \subseteq \mathcal{A}$, $\mathit{Range}(\sigma|_{\mathcal{G}}) \subseteq \mathit{PrologTerms}(\Sigma, \mathcal{A})$, $\sigma|_{\mathcal{A}}$ is not a variable renaming on $\mathcal{A}$, $\mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{N}})) \subseteq \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{A}}))$, $\forall (s, s') \in \mathcal{U} : \forall \sigma'' : (s\sigma|_{\mathcal{G}}\sigma'' = s'\sigma|_{\mathcal{G}}\sigma'' \implies \mathit{Dom}(\sigma'') \not\subseteq \mathcal{N})$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}}))$, $\mathcal{F}' = \mathcal{F} \cup (\mathcal{N}(\mathit{Range}(\sigma|_{\mathcal{F}})) \setminus \mathcal{N}(\mathit{Range}(\sigma|_{\mathcal{N} \setminus (\mathcal{F} \cup \mathcal{N}(H_i))}))) \cup (\mathcal{N}(B_i) \setminus \mathcal{N}(H_i))$, $\sigma' = \mathit{Approx}(\sigma, t, c_i, \mathcal{G}, \mathcal{F})$, and $B'_i = \mathit{Transformed}(B_i, m)$

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

where $i \neq b$, $c_i = H_i \mathbin{:-} B_i$, $\mathit{mgu}(t, H_i) = \sigma$ with $\mathcal{V}(\mathit{Range}(\sigma)) \subseteq \mathcal{V}_{\mathit{fresh}}$, $\sigma|_{\mathcal{A}}$ is a variable renaming on $\mathcal{A}$, $\mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{N}})) \subseteq \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{A}}))$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}}))$, $\mathcal{F}' = \mathcal{F} \cup (\mathcal{N}(\mathit{Range}(\sigma|_{\mathcal{F}})) \setminus \mathcal{N}(\mathit{Range}(\sigma|_{\mathcal{N} \setminus (\mathcal{F} \cup \mathcal{N}(H_i))}))) \cup (\mathcal{N}(B_i) \setminus \mathcal{N}(H_i))$, $\sigma' =$

$Approx(\sigma, t, c_i, \mathcal{G}, \mathcal{F})$, and $B'_i = Transformed(B_i, m)$

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

if there is a μ such that $S = S''\mu$ for a scope variant S'' of S', for all $a \in \mathcal{G}'$, $a\mu \in PrologTerms(\Sigma, \mathcal{G})$, $\mu|_{\mathcal{N}}$ is a variable renaming, $\mathcal{F}'\mu \subseteq \mathcal{F}$, $\mathcal{F}'\mu(Range(\mu|_{\mathcal{A}})) = \varnothing$, and $\mathcal{U}'\mu \subseteq \mathcal{U}$.

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

Here, the *active cuts* $AC(S)$ of a state S are defined as the set of all m such that $S = S' \mid Q, !_m, Q' \mid S''$ or $S = S' \mid (t, Q)^j_m \mid S''$ and $c_j = H_j :\!\text{-}\ B_j, !, B'_j$, while the *active marks* $AM(S)$ of a state S are defined as all m such that $S = S' \mid ?_m \mid S''$ and $S' \neq \varepsilon \neq S''$.

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

where $t' \neq !_m$ for some $m \in \mathbb{N}$, $t' \neq \mathsf{call(x)}$ for some $x \in \mathcal{V}$, $root(t') \in BuiltInPredicates \vee Slice_{\mathcal{P}}(t') \neq \varnothing$, $\delta = ApproxSub(t', \mathcal{G}, \mathcal{F})$, $\mathcal{G}' = \mathcal{G} \cup ApproxGnd(t', \delta)$, and $\mathcal{F}' = \mathcal{F} \setminus \mathcal{F}(t')$.

Here, *ApproxSub* approximates the substitutions of the answer sets of all concretizations w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$ of t':

$$ApproxSub(t', \mathcal{G}, \mathcal{F}) = \begin{cases} \alpha_{\mathcal{F}(t')} & \text{if } \mathcal{V}(t') \subseteq \mathcal{G} \cup \mathcal{F} \\ \alpha_{\mathcal{N}(t')}\alpha_{\mathcal{A}\setminus\mathcal{G}} & \text{if } \mathcal{A}(t') \subseteq \mathcal{G} \wedge \mathcal{N}(t') \not\subseteq \mathcal{F} \\ \alpha_{\mathcal{F}(t')}\alpha_{\mathcal{A}\setminus\mathcal{G}}\alpha_{\mathcal{N}\setminus\mathcal{F}} & \text{otherwise} \end{cases}$$

Finally, *ApproxGnd* approximates the abstract variables that have to be instantiated by ground terms using a given groundness analysis $Ground_{\mathcal{P}} : \Sigma \times 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ which given a predicate p and a set of ground argument positions computes the set of ground arguments positions after a successful computation using the clauses from $\mathcal{P}$:

$$ApproxGnd(t', \delta) =$$

$$\{\mathcal{A}(t_i\delta) \mid t' = p(t_1, \ldots, t_n), i \in Ground_{Slice_{\mathcal{P}}(t')}(p, \{i \mid \mathcal{V}(t_i) \subseteq \mathcal{G}\})\}$$

Approx replaces some variables by fresh abstract variables:

$$Approx(\sigma, t, H_i :\!\text{-}\ B_i, \mathcal{G}, \mathcal{F}) = \begin{cases} \sigma & \text{if } \mathcal{A}(t) \subseteq \mathcal{G} \text{ and } \mathcal{N}(t) \subseteq \mathcal{F} \\ \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'} & \text{if } \mathcal{A}(t) \subseteq \mathcal{G} \text{ and } \mathcal{N}(t) \not\subseteq \mathcal{F} \\ \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')} & \text{if } \mathcal{A}(t) \not\subseteq \mathcal{G} \end{cases}$$

$$\frac{,(t_1,t_2),Q \mid S;KB}{t_1,t_2,Q \mid S;KB}\ (\textsc{Conjunction})$$

$$\frac{;(t_1,t_2),Q \mid S;KB}{t_1,Q \mid t_2,Q \mid S;KB}\ (\textsc{Disjunction})\quad \text{where } root(t_1) \neq \text{->}/2 \text{ and } t_1 \notin \mathcal{A}$$

$$\frac{\mathsf{fail},Q \mid S;KB}{S;KB}\ (\textsc{Fail}) \qquad \frac{\mathsf{halt},Q \mid S;KB}{\varepsilon;KB}\ (\textsc{Halt}) \qquad \frac{\mathsf{halt}(t'),Q \mid S;KB}{\varepsilon;KB}\ (\textsc{Halt1})$$

$$\frac{\text{->}(t_1,t_2),Q \mid S;KB}{\mathsf{call}(t_1),!_m,t_2,Q \mid ?_m \mid S;KB}\ (\textsc{IfThen})\quad \text{for a fresh } m \in \mathbb{N}$$

$$\frac{;(\text{->}(t_1,t_2),t_3),Q \mid S;KB}{\mathsf{call}(t_1),!_m,t_2,Q \mid t_3,Q \mid ?_m \mid S;KB}\ (\textsc{IfThenElse})\quad \text{for a fresh } m \in \mathbb{N}$$

$$\frac{\backslash+(t'),Q \mid S;KB}{\mathsf{call}(t'),!_m,\mathsf{fail} \mid Q \mid ?_m \mid S;KB}\ (\textsc{Not})\quad \text{for a fresh } m \in \mathbb{N}$$

$$\frac{\mathsf{once}(t'),Q \mid S;KB}{\mathsf{call}(,(t',!)),Q \mid S;KB}\ (\textsc{Once}) \qquad \frac{\mathsf{repeat},Q \mid S;KB}{Q \mid \mathsf{repeat},Q \mid S;KB}\ (\textsc{Repeat})$$

$$\frac{\mathsf{throw}(t'),Q \mid S;KB}{\varepsilon;KB}\ (\textsc{Throw}) \qquad \frac{\mathsf{true},Q \mid S;KB}{Q \mid S;KB}\ (\textsc{True})$$

$$\frac{==(t_1,t_1),Q \mid S;KB}{Q \mid S;KB}\ (\textsc{EqualsSuccess})$$

$$\frac{==(t_1,t_2),Q \mid S;KB}{S;KB}\ (\textsc{EqualsFail})\quad \begin{array}{l}\text{where } t_1 \nsim t_2 \text{ or } \forall\sigma \text{ with } t_1\sigma = t_2\sigma \text{ we}\\ \text{have } Dom(\sigma) \cap \mathcal{N} \neq \varnothing\end{array}$$

$$\frac{==(t_1,t_2),Q \mid S;(\mathcal{G},\mathcal{F},\mathcal{U})}{Q\sigma,\mid S\sigma;(\mathcal{G}',\mathcal{F},\mathcal{U}') \qquad S;(\mathcal{G},\mathcal{F},\mathcal{U})}\ (\textsc{EqualsCase})$$

where $t_1 \neq t_2$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$, $\mathcal{U}' = \mathcal{U}\sigma$ and $mgu(t_1,t_2) = \sigma$ with $Dom(\sigma) \subseteq \mathcal{A}$ and $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{A}_{fresh}$

$$\frac{\backslash==(t_1,t_2),Q \mid S;KB}{Q \mid S;KB}\ (\textsc{UnequalsSuccess})\quad \begin{array}{l}\text{where } t_1 \nsim t_2 \text{ or } \forall\sigma\\ \text{with } t_1\sigma = t_2\sigma \text{ we have}\\ Dom(\sigma) \cap \mathcal{N} \neq \varnothing\end{array}$$

$$\frac{\backslash==(t_1,t_1),Q \mid S;KB}{S;KB}\ (\textsc{UnequalsFail})$$

$$\frac{\backslash==(t_1,t_2),Q \mid S;(\mathcal{G},\mathcal{F},\mathcal{U})}{Q,\mid S;(\mathcal{G},\mathcal{F},\mathcal{U}) \qquad S\sigma;(\mathcal{G}',\mathcal{F},\mathcal{U}')}\ (\textsc{UnequalsCase})$$

where $t_1 \neq t_2$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_\mathcal{G}))$, $\mathcal{U}' = \mathcal{U}\sigma$ and $mgu(t_1, t_2) = \sigma$ with $Dom(\sigma) \subseteq \mathcal{A}$ and $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{A}_{fresh}$

$$\frac{=(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q\sigma' \mid S\sigma|_\mathcal{G}; (\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_\mathcal{G})} \text{ (UnifySuccess)}$$

where $mgu(t_1, t_2) = \sigma$ with $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $Range(\sigma|_\mathcal{A}) \subseteq \mathcal{A}$, $\sigma|_\mathcal{A} : Dom(\sigma|_\mathcal{A}) \to Range(\sigma|_\mathcal{A})$ is bijective, $\mathcal{A}(Range(\sigma|_\mathcal{N})) \subseteq \mathcal{A}(Range(\sigma|_\mathcal{A}))$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_\mathcal{G}))$, $\mathcal{F}' = \mathcal{F} \cup (\mathcal{N}(Range(\sigma|_\mathcal{F})) \setminus \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus\mathcal{F}})))$ and $\sigma' = ApproxUnify(\sigma, t_1, t_2, \mathcal{G}, \mathcal{F})$

$$\frac{=(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{S; (\mathcal{G}, \mathcal{F}, \mathcal{U} \cup \{(t_1, t_2)\})} \text{ (UnifyFail)}$$

where $t_1 \nsim t_2$ or $\sigma = mgu(t_1, t_2)$ with $\exists a \in \mathcal{G} : a\sigma \notin PrologTerms(\Sigma, \mathcal{V})$ or $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $\mathcal{V}(Range(\sigma|_\mathcal{G})) \subseteq \mathcal{A}$ and $\exists (s, s') \in \mathcal{U} : \sigma' = mgu(s\sigma|_\mathcal{G}, s'\sigma|_\mathcal{G}) \wedge Dom(\sigma') \subseteq \mathcal{F}$

$$\frac{=(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q\sigma' \mid S\sigma|_\mathcal{G}; (\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_\mathcal{G}) \qquad S; (\mathcal{G}, \mathcal{F}, \mathcal{U} \cup \{(t_1, t_2)\})} \text{ (UnifyCase)}$$

where $mgu(t_1, t_2) = \sigma$ with $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $\mathcal{V}(Range(\sigma|_\mathcal{A})) \subseteq \mathcal{A}$, $Range(\sigma|_\mathcal{G}) \subseteq PrologTerms(\Sigma, \mathcal{A})$, $(Range(\sigma|_\mathcal{A}) \not\subseteq \mathcal{A} \vee \sigma|_\mathcal{A} : Dom(\sigma|_\mathcal{A}) \to Range(\sigma|_\mathcal{A})$ is not bijective), $\mathcal{A}(Range(\sigma|_\mathcal{N})) \subseteq \mathcal{A}(Range(\sigma|_\mathcal{A}))$, $\forall (s, s') \in \mathcal{U} : \forall \sigma'' : (s\sigma|_\mathcal{G}\sigma'' = s'\sigma|_\mathcal{G}\sigma'' \implies Dom(\sigma'') \not\subseteq \mathcal{F})$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_\mathcal{G}))$, $\mathcal{F}' = \mathcal{F} \cup (\mathcal{N}(Range(\sigma|_\mathcal{F})) \setminus \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus\mathcal{F}})))$ and $\sigma' = ApproxUnify(\sigma, t_1, t_2, \mathcal{G}, \mathcal{F})$

$$\frac{\setminus=(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U} \cup \{(t_1, t_2)\})} \text{ (NoUnifySuccess)}$$

where $t_1 \nsim t_2$ or $\sigma = mgu(t_1, t_2)$ with $\exists a \in \mathcal{G} : a\sigma \notin PrologTerms(\Sigma, \mathcal{V})$ or $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $\mathcal{V}(Range(\sigma|_\mathcal{G})) \subseteq \mathcal{A}$ and $\exists (s, s') \in \mathcal{U} : \sigma' = mgu(s\sigma|_\mathcal{G}, s'\sigma|_\mathcal{G}) \wedge Dom(\sigma') \subseteq \mathcal{F}$

$$\frac{\setminus=(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{S\sigma|_\mathcal{G}; (\mathcal{G}', \mathcal{F}, \mathcal{U}\sigma|_\mathcal{G})} \text{ (NoUnifyFail)}$$

where $mgu(t_1, t_2) = \sigma$ with $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $Range(\sigma|_\mathcal{A}) \subseteq \mathcal{A}$, $\sigma|_\mathcal{A} : Dom(\sigma|_\mathcal{A}) \to Range(\sigma|_\mathcal{A})$ is bijective, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_\mathcal{G}))$ and $\mathcal{A}(Range(\sigma|_\mathcal{N})) \subseteq \mathcal{A}(Range(\sigma|_\mathcal{A}))$

$$\frac{\setminus=(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U} \cup \{(t_1, t_2)\}) \qquad S\sigma|_\mathcal{G}; (\mathcal{G}', \mathcal{F}, \mathcal{U}\sigma|_\mathcal{G})} \text{ (NoUnifyCase)}$$

where $mgu(t_1, t_2) = \sigma$ with $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $\mathcal{V}(Range(\sigma|_\mathcal{A})) \subseteq \mathcal{A}$, $Range(\sigma|_\mathcal{G}) \subseteq PrologTerms(\Sigma, \mathcal{A})$, $(Range(\sigma|_\mathcal{A}) \not\subseteq \mathcal{A} \vee \sigma|_\mathcal{A} : Dom(\sigma|_\mathcal{A}) \to Range(\sigma|_\mathcal{A})$ is not bijective), $\mathcal{A}(Range(\sigma|_\mathcal{N})) \subseteq \mathcal{A}(Range(\sigma|_\mathcal{A}))$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_\mathcal{G}))$ and $\forall (s, s') \in \mathcal{U} : \forall \sigma'' : (s\sigma|_\mathcal{G}\sigma'' = s'\sigma|_\mathcal{G}\sigma'' \implies Dom(\sigma'') \not\subseteq \mathcal{F})$

ApproxUnify replaces some variables by fresh abstract variables:

$$ApproxUnify(\sigma, t_1, t_2, \mathcal{G}, \mathcal{F}) = \begin{cases} \sigma & \text{if } \mathcal{A}(t_1) \cup \mathcal{A}(t_2) \subseteq \mathcal{G} \\ & \text{and } \mathcal{N}(t_1) \cup \mathcal{N}(t_2) \subseteq \mathcal{F} \\ \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'} & \text{if } \mathcal{A}(t_1) \cup \mathcal{A}(t_2) \subseteq \mathcal{G} \\ & \text{and } \mathcal{N}(t_1) \cup \mathcal{N}(t_2) \not\subseteq \mathcal{F} \\ \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')} & \text{if } \mathcal{A}(t_1) \cup \mathcal{A}(t_2) \not\subseteq \mathcal{G} \end{cases}$$

$$\frac{\mathsf{atomic}(c), Q \mid S; KB}{Q \mid S; KB}\ (\textsc{AtomicSuccess}) \quad \text{where } c \text{ is a constant}$$

$$\frac{\mathsf{atomic}(t'), Q \mid S; KB}{S; KB}\ (\textsc{AtomicFail}) \quad \text{where } t' \text{ is no constant and no abstract variable}$$

$$\frac{\mathsf{atomic}(a), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q \mid S; (\mathcal{G} \cup \{a\}, \mathcal{F}, \mathcal{U}) \qquad S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}\ (\textsc{AtomicCase}) \quad \text{where } a \in \mathcal{A}$$

$$\frac{\mathsf{compound}(f(t_1, \ldots, t_k)), Q \mid S; KB}{Q \mid S; KB}\ (\textsc{CompoundSuccess}) \quad \begin{array}{l}\text{where } f/k \in \Sigma \\ \text{and } t_i \in \\ \mathit{PrologTerms}(\Sigma, \mathcal{V}) \\ \text{for all } i \in \{1, \ldots, k\}\end{array}$$

$$\frac{\mathsf{compound}(t'), Q \mid S; KB}{S; KB}\ (\textsc{CompoundFail}) \quad \text{where } t' \text{ is a constant or } t' \in \mathcal{N}$$

$$\frac{\mathsf{compound}(a), Q \mid S; KB}{Q \mid S; KB \qquad S; KB}\ (\textsc{CompoundCase}) \quad \text{where } a \in \mathcal{A}$$

$$\frac{\mathsf{nonvar}(t'), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}\ (\textsc{NonvarSuccess}) \quad \text{where } t' \notin \mathcal{V} \setminus \mathcal{G}$$

$$\frac{\mathsf{nonvar}(x), Q \mid S; KB}{S; KB}\ (\textsc{NonvarFail})$$

$$\frac{\mathsf{nonvar}(a), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}) \qquad S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}\ (\textsc{NonvarCase}) \quad \text{where } a \in \mathcal{A} \setminus \mathcal{G}$$

$$\frac{\mathsf{var}(x), Q \mid S; KB}{Q \mid S; KB}\ (\textsc{VarSuccess})$$

$$\frac{\mathsf{var}(t'), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}\ (\textsc{VarFail}) \quad \text{where } t' \notin \mathcal{V} \setminus \mathcal{G}$$

$$\frac{\mathsf{flush\_output}, Q \mid S; KB}{Q \mid S; KB}\ (\textsc{FlushOutput}) \qquad \frac{\mathsf{nl}, Q \mid S; KB}{Q \mid S; KB}\ (\textsc{Newline})$$

$$\frac{\mathsf{write}(t'), Q \mid S; KB}{Q \mid S; KB}\ (\textsc{Write}) \qquad \frac{\mathsf{write\_canonical}(t'), Q \mid S; KB}{Q \mid S; KB}\ (\textsc{WriteCanonical})$$

$$\frac{\mathsf{writeq}(t'), Q \mid S; KB}{Q \mid S; KB}\ (\textsc{Writeq})$$

For a graph G and an abstract inference rule Rule, we use the notation $Rule(G)$ to denote all nodes of G to which Rule has been applied. We denote by $Succ(i, n)$ the i-th child of n and by $Succ(i, Rule(G))$ the set of i-th children of nodes from $Rule(G)$.

For identifying different states where the only difference lies in a scope renaming, we introduce the notion of a scope variant.

Definition A 6 (*Scope Variant (Ströder 2010)*)
Given a concrete (abstract) state S, we call a concrete (abstract) state S' a *scope variant* of S, iff there is a bijection $f : \mathbb{N} \to \mathbb{N}$, both states have the same length and the following conditions are satisfied for all $i \in \{1, \ldots, length(S)\}$ and elements e_i of S at position i and e'_i of S' at position i:

- If e_i is an unlabeled list of terms t, then e'_i is an unlabeled list of terms t' with $t' = t[!_j/!_{f(j)} \forall j \in \mathbb{N}]$.
- If e_i is a labeled list of terms t^r_s, then e'_i is a labeled list of terms $t'^r_{f(s)}$ with $t' = t[!_j/!_{f(j)} \forall j \in \mathbb{N}]$.
- If $e_i = ?_s$, then $e'_i = ?_{f(s)}$.

Lemma A 7 (*Equivalent Evaluations for Concrete Scope Variants (Ströder 2010)*)

Given a concrete state S and a scope variant S' of S, all evaluations possible for S are also possible for S'.

Proof
To show Lemma A 7 it is sufficient to show that for all concrete rules the applicability of a rule for S implies the applicability for S' and after application of the rule the resulting states are still scope variants of each other. We perform a case analysis over the applicability of the concrete inference rules for S.

- Success is applicable:
Then we have $S = \square \mid S''$. Since S' is a scope variant of S, we also have $S' = \square \mid S'''$ and Success is applicable for S', too. After application of Success we obtain the states S'' and S''', which are scope variants of each other as $\square \mid S''$ and $\square \mid S'''$ are scope variants.
- Failure is applicable:
Then we have $S = ?_s \mid S''$ and as S' is a scope variant of S, we also have $S' = ?_{f(s)} \mid S'''$. Thus, Failure is applicable for S', too. After application of Failure

we obtain the states S'' and S''', which are scope variants of each other as $?_s \mid S''$ and $?_{f(s)} \mid S'''$ are scope variants.

• VariableError is applicable:
Then we have $S = \mathsf{call}(x), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{call}(x), Q' \mid S'''$. Thus, VariableError is applicable for S', too. After application of VariableError we obtain the states ε and ε, which clearly are scope variants of each other.

• UndefinedError is applicable:
Then we have $S = t, Q \mid S''$ where $\mathit{Slice}_{\mathcal{P}}(t) = \varnothing$ and as S' is a scope variant of S, we also have $S' = t', Q' \mid S'''$ with $\mathit{Slice}_{\mathcal{P}}(t') = \varnothing$. Thus, UndefinedError is applicable for S', too. After application of VariableError we obtain the states ε and ε, which clearly are scope variants of each other.

• Cut is applicable:
Then we have $S = !_s, Q \mid S'' \mid ?_s \mid S'''$ with S'' contains no $?_s$ and as S' is a scope variant of S, we also have $S' = !_{f(s)}, Q' \mid S'''' \mid ?_{f(s)} \mid S''''$ with S'''' contains no $?_{f(s)}$. Thus, Cut is applicable for S', too. After application of Cut we obtain the states $Q \mid ?_s \mid S'''$ and $Q' \mid ?_{f(s)} \mid S''''$, which are scope variants of each other as $!_s, Q \mid S'' \mid ?_s \mid S'''$ and $!_{f(s)}, Q' \mid S'''' \mid ?_{f(s)} \mid S''''$ are scope variants.

• CutAll is applicable:
Then we have $S = !_s, Q \mid S''$ with S'' contains no $?_s$ and as S' is a scope variant of S, we also have $S' = !_{f(s)}, Q' \mid S'''$ with S''' contains no $?_{f(s)}$. Thus, CutAll is applicable for S', too. After application of CutAll we obtain the states Q and Q', which are scope variants of each other as $!_s, Q \mid S''$ and $!_{f(s)}, Q' \mid S'''$ are scope variants.

• Case is applicable:
Then we have $S = t, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = t', Q' \mid S'''$. Thus, Case is applicable for S', too. After application of Case we obtain the states $(t, Q)^{i_1}_m \mid \ldots \mid (t, Q)^{i_k}_m \mid ?_m \mid S''$ and $(t', Q')^{i'_1}_n \mid \ldots \mid (t', Q')^{i'_{k'}}_n \mid ?_n \mid S'''$, where m and n are fresh, $i_1 < \ldots < i_k$, $i'_1 < \ldots < i'_{k'}$, $\mathit{Slice}_{\mathcal{P}}(t) = \{c_{i_1}, \ldots, c_{i_k}\}$ and $\mathit{Slice}_{\mathcal{P}}(t') = \{c_{i'_1}, \ldots, c_{i'_{k'}}\}$. As S and S' are scope variants, t and t' have the same root symbol and, thus, we have $\mathit{Slice}_{\mathcal{P}}(t) = \mathit{Slice}_{\mathcal{P}}(t')$. W.l.o.g. we can also demand $f(m) = n$ as both m and n are fresh. Hence, the second state is $(t', Q')^{i_1}_{f(m)} \mid \ldots \mid (t', Q')^{i_k}_{f(m)} \mid ?_{f(m)} \mid S'''$, which is a scope variant of $(t, Q)^{i_1}_m \mid \ldots \mid (t, Q)^{i_k}_m \mid ?_m \mid S''$ as $t, Q \mid S''$ and $t', Q' \mid S'''$ are scope variants.

• Eval is applicable:
Then we have $S = (t, Q)^i_m \mid S''$ with $c_i = H_i \mathbin{:\!-} B_i$ and $\mathit{mgu}(t, H_i) = \sigma$ and as S' is a scope variant of S, we also have $S' = (t', Q')^i_{f(m)} \mid S'''$ with $\mathit{mgu}(t', H_i) = \sigma'$. Thus, Eval is applicable for S', too. After application of Eval we obtain the states $B'_i\sigma, Q\sigma \mid S''$ and $B''_i\sigma', Q'\sigma' \mid S'''$, where $B'_i = \mathit{Transformed}(B_i, m)$ and $B''_i = \mathit{Transformed}(B_i, f(m))$. As S and S' are scope variants, we have for all terms $r \in \mathit{Range}(\sigma)$ and $r' \in \mathit{Range}(\sigma')$ that $r' = r[!_j/!_{f(j)} \forall j \in \mathbb{N}]$ and $\mathit{Dom}(\sigma) = \mathit{Dom}(\sigma')$. Hence, $B'_i\sigma, Q\sigma \mid S''$ and $B''_i\sigma', Q'\sigma' \mid S'''$ are scope variants of each other as $(t, Q)^i_m \mid S''$ and $(t', Q')^i_{f(m)} \mid S'''$ are scope variants.

• Backtrack is applicable:
Then we have $S = (t, Q)^i_m \mid S''$ where $c_i = H_i \mathbin{:\!-} B_i$ and $t \nsim H_i$. As S' is a scope

variant of S, we also have $S' = (t', Q')^i_{f(m)} \mid S'''$ where $t' \nsim H_i$. Thus, BACKTRACK is applicable for S', too. After application of BACKTRACK we obtain the states S'' and S''', which are scope variants of each other as $(t, Q)^i_m \mid S''$ and $(t', Q')^i_{f(m)} \mid S'''$ are scope variants.

• CALL is applicable:
Then we have $S = \mathsf{call}(t'), Q \mid S''$ where $t' \in \mathit{PrologTerms}(\Sigma, \mathcal{V}) \setminus \mathcal{V}$ and t' has only finitely many predication positions. As S' is a scope variant of S, we also have $S' = \mathsf{call}(t''), Q' \mid S'''$ where $t'' \in \mathit{PrologTerms}(\Sigma, \mathcal{V}) \setminus \mathcal{V}$ and t'' has only finitely many predication positions. Thus, CALL is applicable for S', too. After application of CALL we obtain the states $t''', Q \mid ?_m \mid S''$ and $t'''', Q' \mid ?_{m'} \mid S'''$ where $t''' = \mathit{Transformed}(t', m)$ and $t'''' = \mathit{Transformed}(t'', m')$. As m and m' are fresh, we can demand $m' = f(m)$. Since the transformation by the function *Transformed* uses the same scope for all cuts in predication positions, the reached states are scope variants of each other as $\mathsf{call}(t'), Q \mid S''$ and $\mathsf{call}(t''), Q' \mid S'''$ are scope variants.

• ATOMICFAIL is applicable:
Then we have $S = \mathsf{atomic}(t'), Q \mid S''$ with t' not being a constant and as S' is a scope variant of S, we also have $S' = \mathsf{atomic}(t''), Q' \mid S'''$ with t'' not being a constant. Thus, ATOMICFAIL is applicable for S', too. After application of ATOMICFAIL we obtain the states S'' and S''' which are scope variants of each other as $\mathsf{atomic}(t'), Q \mid S''$ and $\mathsf{atomic}(t''), Q' \mid S'''$ are scope variants.

• ATOMICSUCCESS is applicable:
Then we have $S = \mathsf{atomic}(c), Q \mid S''$ with c being a constant and as S' is a scope variant of S, we also have $S' = \mathsf{atomic}(c), Q' \mid S'''$. Thus, ATOMICSUCCESS is applicable for S', too. After application of ATOMICSUCCESS we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{atomic}(c), Q \mid S''$ and $\mathsf{atomic}(c), Q' \mid S'''$ are scope variants.

• COMPOUNDFAIL is applicable:
Then we have $S = \mathsf{compound}(t'), Q \mid S''$ with t' being a constant or a variable and as S' is a scope variant of S, we also have $S' = \mathsf{compound}(t'), Q' \mid S'''$. Thus, COMPOUNDFAIL is applicable for S', too. After application of COMPOUNDFAIL we obtain the states S'' and S''' which are scope variants of each other as $\mathsf{compound}(t'), Q \mid S''$ and $\mathsf{compound}(t'), Q' \mid S'''$ are scope variants.

• COMPOUNDSUCCESS is applicable:
Then we have $S = \mathsf{compound}(t'), Q \mid S''$ with t' not being a constant or variable and as S' is a scope variant of S, we also have $S' = \mathsf{compound}(t''), Q' \mid S'''$ with t'' not being a constant or variable. Thus, COMPOUNDSUCCESS is applicable for S', too. After application of COMPOUNDSUCCESS we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{compound}(t'), Q \mid S''$ and $\mathsf{compound}(t''), Q' \mid S'''$ are scope variants.

• CONJUNCTION is applicable:
Then we have $S = ,(t_1, t_2), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = ,(t'_1, t'_2), Q' \mid S'''$. Thus, CONJUNCTION is applicable for S', too. After application of CONJUNCTION we obtain the states $t_1, t_2, Q \mid S''$ and $t'_1, t'_2, Q' \mid S'''$

which are scope variants of each other as $,(t_1, t_2), Q \mid S''$ and $,(t'_1, t'_2), Q' \mid S'''$ are scope variants.

- DISJUNCTION is applicable:
Then we have $S = ;(t_1, t_2), Q \mid S''$ where $root(t_1) \neq \text{->}/2$ and as S' is a scope variant of S, we also have $S' = ;(t'_1, t'_2), Q' \mid S'''$ where $root(t'_1) \neq \text{->}/2$. Thus, DISJUNCTION is applicable for S', too. After application of DISJUNCTION we obtain the states $t_1, Q \mid t_2, Q \mid S''$ and $t'_1, Q' \mid t'_2, Q' \mid S'''$ which are scope variants of each other as $;(t_1, t_2), Q \mid S''$ and $;(t'_1, t'_2), Q' \mid S'''$ are scope variants.
- EQUALSFAIL is applicable:
Then we have $S = ==(t_1, t_2), Q \mid S''$ with $t_1 \neq t_2$ and as S' is a scope variant of S, we also have $S' = ==(t'_1, t'_2), Q' \mid S'''$ with $t'_1 \neq t'_2$. Thus, EQUALSFAIL is applicable for S', too. After application of EQUALSFAIL we obtain the states S'' and S''' which are scope variants of each other as $==(t_1, t_2), Q \mid S''$ and $==(t'_1, t'_2), Q' \mid S'''$ are scope variants.
- EQUALSSUCCESS is applicable:
Then we have $S = ==(t_1, t_1), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = ==(t'_1, t'_1), Q' \mid S'''$. Thus, EQUALSSUCCESS is applicable for S', too. After application of EQUALSSUCCESS we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $==(t_1, t_1), Q \mid S''$ and $==(t'_1, t'_1), Q' \mid S'''$ are scope variants.
- FAIL is applicable:
Then we have $S = \mathsf{fail}, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{fail}, Q' \mid S'''$. Thus, FAIL is applicable for S', too. After application of FAIL we obtain the states S'' and S''' which are scope variants of each other as $\mathsf{fail}, Q \mid S''$ and $\mathsf{fail}, Q' \mid S'''$ are scope variants.
- FLUSHOUTPUT is applicable:
Then we have $S = \mathsf{flush\_output}, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{flush\_output}, Q' \mid S'''$. Thus, FLUSHOUTPUT is applicable for S', too. After application of FLUSHOUTPUT we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{flush\_output}, Q \mid S''$ and $\mathsf{flush\_output}, Q' \mid S'''$ are scope variants.
- HALT is applicable:
Then we have $S = \mathsf{halt}, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{halt}, Q' \mid S'''$. Thus, HALT is applicable for S', too. After application of HALT we obtain the states ε and ε which clearly are scope variants of each other.
- HALT1 is applicable:
Then we have $S = \mathsf{halt}(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{halt}(t''), Q' \mid S'''$. Thus, HALT1 is applicable for S', too. After application of HALT1 we obtain the states ε and ε which clearly are scope variants of each other.
- IFTHEN is applicable:
Then we have $S = \text{->}(t_1, t_2), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \text{->}(t'_1, t'_2), Q' \mid S'''$. Thus, IFTHEN is applicable for S', too. After application of IFTHEN we obtain the states $\mathsf{call}(t_1), !_m, t_2, Q \mid ?_m \mid S''$ and $\mathsf{call}(t'_1), !_{m'}, t'_2, Q' \mid ?_{m'} \mid S'''$. As m and m' are fresh, we can demand that $m' = f(m)$. So the reached

states are scope variants of each other as $\text{->}(t_1, t_2), Q \mid S''$ and $\text{->}(t'_1, t'_2), Q' \mid S'''$ are scope variants.

• IfThenElse is applicable:
Then we have $S = ;(\text{->}(t_1, t_2), t_3), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = ;(\text{->}(t'_1, t'_2), t'_3), Q' \mid S'''$. Thus, IfThenElse is applicable for S', too. After application of IfThenElse we obtain the states $\mathsf{call}(t_1), !_m, t_2, Q \mid t_3, Q \mid ?_m \mid S''$ and $\mathsf{call}(t'_1), !_{m'}, t'_2, Q' \mid t'_3, Q' \mid ?_{m'} \mid S'''$. As m and m' are fresh, we can demand that $m' = f(m)$. So the reached states are scope variants of each other as $;(\text{->}(t_1, t_2), t_3), Q \mid S''$ and $;(\text{->}(t'_1, t'_2), t'_3), Q' \mid S'''$ are scope variants.

• Newline is applicable:
Then we have $S = \mathsf{nl}, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{nl}, Q' \mid S'''$. Thus, Newline is applicable for S', too. After application of Newline we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{nl}, Q \mid S''$ and $\mathsf{nl}, Q' \mid S'''$ are scope variants.

• NonvarFail is applicable:
Then we have $S = \mathsf{nonvar}(x), Q \mid S''$ with $x \in \mathcal{N}$ and as S' is a scope variant of S, we also have $S' = \mathsf{nonvar}(x), Q' \mid S'''$. Thus, NonvarFail is applicable for S', too. After application of NonvarFail we obtain the states S'' and S''' which are scope variants of each other as $\mathsf{nonvar}(x), Q \mid S''$ and $\mathsf{nonvar}(x), Q' \mid S'''$ are scope variants.

• NonvarSuccess is applicable:
Then we have $S = \mathsf{nonvar}(t'), Q \mid S''$ with t' not being a variable and as S' is a scope variant of S, we also have $S' = \mathsf{nonvar}(t''), Q' \mid S'''$ with t'' not being a variable. Thus, NonvarSuccess is applicable for S', too. After application of NonvarSuccess we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{nonvar}(t'), Q \mid S''$ and $\mathsf{nonvar}(t''), Q' \mid S'''$ are scope variants.

• Not is applicable:
Then we have $S = \backslash+(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \backslash+(t''), Q' \mid S'''$. Thus, Not is applicable for S', too. After application of Not we obtain the states $\mathsf{call}(t'), !_m, \mathsf{fail} \mid Q \mid ?_m \mid S''$ and $\mathsf{call}(t''), !_{m'}, \mathsf{fail} \mid Q' \mid ?_{m'} \mid S'''$. As m and m' are fresh, we can demand that $m' = f(m)$. So the reached states are scope variants of each other as $\backslash+(t'), Q \mid S''$ and $\backslash+(t''), Q' \mid S'''$ are scope variants.

• NoUnifyFail is applicable:
Then we have $S = \backslash=(t_1, t_2), Q \mid S''$ where $t_1 \sim t_2$ and as S' is a scope variant of S, we also have $S' = \backslash=(t'_1, t'_2), Q' \mid S'''$ where $t'_1 \sim t'_2$. Thus, NoUnifyFail is applicable for S', too. After application of NoUnifyFail we obtain the states S'' and S''' which are scope variants of each other as $\backslash=(t_1, t_2), Q \mid S''$ and $\backslash=(t'_1, t'_2), Q' \mid S'''$ are scope variants.

• NoUnifySuccess is applicable:
Then we have $S = \backslash=(t_1, t_2), Q \mid S''$ where $t_1 \nsim t_2$ and as S' is a scope variant of S, we also have $S' = \backslash=(t'_1, t'_2), Q' \mid S'''$ where $t'_1 \nsim t'_2$. Thus, NoUnifySuccess is applicable for S', too. After application of NoUnifySuccess we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\backslash=(t_1, t_2), Q \mid S''$ and $\backslash=(t'_1, t'_2), Q' \mid S'''$ are scope variants.

• Once is applicable:
Then we have $S = \mathsf{once}(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{once}(t''), Q' \mid S'''$. Thus, Once is applicable for S', too. After application of Once we obtain the states $\mathsf{call}(,(t', !)), Q \mid S''$ and $\mathsf{call}(,(t'', !)), Q' \mid S'''$ which are scope variants of each other as $\mathsf{once}(t'), Q \mid S''$ and $\mathsf{once}(t''), Q' \mid S'''$ are scope variants.
• Repeat is applicable:
Then we have $S = \mathsf{repeat}, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{repeat}, Q' \mid S'''$. Thus, Repeat is applicable for S', too. After application of Repeat we obtain the states $Q \mid \mathsf{repeat}, Q \mid S''$ and $Q' \mid \mathsf{repeat}, Q' \mid S'''$ which are scope variants of each other as $\mathsf{repeat}, Q \mid S''$ and $\mathsf{repeat}, Q' \mid S'''$ are scope variants.
• Throw is applicable:
Then we have $S = \mathsf{throw}(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{throw}(t''), Q' \mid S'''$. Thus, Throw is applicable for S', too. After application of Throw we obtain the states ε and ε which clearly are scope variants of each other.
• True is applicable:
Then we have $S = \mathsf{true}, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{true}, Q' \mid S'''$. Thus, True is applicable for S', too. After application of True we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{true}, Q \mid S''$ and $\mathsf{true}, Q' \mid S'''$ are scope variants.
• UnequalsFail is applicable:
Then we have $S = \backslash\!\!=\!\!=(t_1, t_1), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \backslash\!\!=\!\!=(t'_1, t'_1), Q' \mid S'''$. Thus, UnequalsFail is applicable for S', too. After application of UnequalsFail we obtain the states S'' and S''' which are scope variants of each other as $\backslash\!\!=\!\!=(t_1, t_1), Q \mid S''$ and $\backslash\!\!=\!\!=(t'_1, t'_1), Q' \mid S'''$ are scope variants.
• UnequalsSuccess is applicable:
Then we have $S = \backslash\!\!=\!\!=(t_1, t_2), Q \mid S''$ where $t_1 \neq t_2$ and as S' is a scope variant of S, we also have $S' = \backslash\!\!=\!\!=(t'_1, t'_2), Q' \mid S'''$ where $t'_1 \neq t'_2$. Thus, UnequalsSuccess is applicable for S', too. After application of UnequalsSuccess we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\backslash\!\!=\!\!=(t_1, t_2), Q \mid S''$ and $\backslash\!\!=\!\!=(t'_1, t'_2), Q' \mid S'''$ are scope variants.
• UnifyFail is applicable:
Then we have $S = =(t_1, t_2), Q \mid S''$ with $t_1 \nsim t_2$ and as S' is a scope variant of S, we also have $S' = =(t'_1, t'_2), Q' \mid S'''$ with $t'_1 \nsim t'_2$. Thus, UnifyFail is applicable for S', too. After application of UnifyFail we obtain the states S'' and S''' which are scope variants of each other as $=(t_1, t_2), Q \mid S''$ and $=(t'_1, t'_2), Q' \mid S'''$ are scope variants.
• UnifySuccess is applicable:
Then we have $S = =(t_1, t_2), Q \mid S''$ where $t_1 \sim t_2$ and as S' is a scope variant of S, we also have $S' = =(t'_1, t'_2), Q' \mid S'''$ where $t'_1 \sim t'_2$. Thus, UnifySuccess is applicable for S', too. After application of UnifySuccess we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $=(t_1, t_2), Q \mid S''$ and $=(t'_1, t'_2), Q' \mid S'''$ are scope variants.

• VarFail is applicable:
Then we have $S = \mathsf{var}(t'), Q \mid S''$ with t' not being a variable and as S' is a scope variant of S, we also have $S' = \mathsf{var}(t''), Q' \mid S'''$ with t'' not being a variable. Thus, VarFail is applicable for S', too. After application of VarFail we obtain the states S'' and S''' which are scope variants of each other as $\mathsf{var}(t'), Q \mid S''$ and $\mathsf{var}(t''), Q' \mid S'''$ are scope variants.
• VarSuccess is applicable:
Then we have $S = \mathsf{var}(x), Q \mid S''$ with $x \in \mathcal{N}$ and as S' is a scope variant of S, we also have $S' = \mathsf{var}(x), Q' \mid S'''$. Thus, VarSuccess is applicable for S', too. After application of VarSuccess we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{var}(x), Q \mid S''$ and $\mathsf{var}(x), Q' \mid S'''$ are scope variants.
• Write is applicable:
Then we have $S = \mathsf{write}(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{write}(t''), Q' \mid S'''$. Thus, Write is applicable for S', too. After application of Write we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{write}(t'), Q \mid S''$ and $\mathsf{write}(t''), Q' \mid S'''$ are scope variants.
• WriteCanonical is applicable:
Then we have $S = \mathsf{write\_canonical}(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{write\_canonical}(t''), Q' \mid S'''$. Thus, WriteCanonical is applicable for S', too. After application of WriteCanonical we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{write\_canonical}(t'), Q \mid S''$ and $\mathsf{write\_canonical}(t''), Q' \mid S'''$ are scope variants.
• Writeq is applicable:
Then we have $S = \mathsf{writeq}(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{writeq}(t''), Q' \mid S'''$. Thus, Writeq is applicable for S', too. After application of Writeq we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{writeq}(t'), Q \mid S''$ and $\mathsf{writeq}(t''), Q' \mid S'''$ are scope variants.

□

Lemma A 8 (*Equivalent Evaluations for Abstract Scope Variants (Ströder 2010)*)
Given an abstract state S and a scope variant S' of S, for every concrete state S_c represented by S there exists a concrete state S'_c represented by S' such that all evaluations possible for S_c are also possible for S'_c.

Proof
As concretizations only replace abstract variables, we have for every concretization γ that $S'\gamma$ is a scope variant of $S\gamma$. By Lemma A 7 we obtain that all evaluations possible for $S\gamma$ are also possible for $S'\gamma$. □

We now state some of the soundness proofs for the abstract inference rules which are referenced later in the proofs for the correctness of our transformation.

Lemma A 9 (*Soundness of* Parallel*, cf. (Schneider-Kamp 2008)*)
The rule Parallel is sound. Moreover, each derivation for a concrete state represented by the abstract state, where Parallel is applied to, either reaches a concrete

state represented by the abstract state's second successor or drops all goals (except at most one scope marker) in this second successor due to a cut before they are evaluated.

Proof
Assume that $S\gamma \mid S'\gamma \in \mathcal{CON}(S \mid S'; KB)$ has an infinite derivation. Then there are three cases. If $S\gamma$ has an infinite derivation, we immediately have that $S\gamma \in \mathcal{CON}(S; KB)$ has an infinite derivation. If $S\gamma$ does not have an infinite derivation and, after finitely many steps, we reach the state $S'\gamma$, we have that $S'\gamma \in \mathcal{CON}(S'; KB)$ has an infinite derivation. Finally, if $S\gamma$ has no infinite derivation, but we do not reach $S'\gamma$, S' must be of the form $S'' \mid ?_m \mid S'''$ with $S'' \neq \varepsilon$ and in the 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 derivation, we get $S''' \neq \varepsilon$. But $S'' \neq \varepsilon \neq S'''$ implies $m \in AM(S')$. Thus we have a contradiction to $AC(S) \cap AM(S') = \varnothing$. □

Lemma A 10 (*Soundness of* SPLIT *(Ströder 2010)*)
The rule SPLIT is sound. Additionally, for every concretization γ w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$ and for every answer substitution δ' of a successful evaluation for $t\gamma$, there is a concretization γ' w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\delta)$ such that $\gamma\delta' = \delta\gamma'$ 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 $t'\gamma, Q\gamma \in \mathcal{CON}(t', Q; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ has an infinite evaluation. Then, following the proof in (Schneider-Kamp 2008), there are two cases. If $t'\gamma$ has an infinite evaluation, we immediately have that $t'\gamma \in \mathcal{CON}(t'; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ has an infinite evaluation. If $t'\gamma$ does not have an infinite evaluation and we did not reach a state of the form $Q\gamma\delta' \mid S'\gamma$ for some answer substitution δ' and state S', we would reach the state ε, which contradicts our assumption that $t'\gamma, Q\gamma$ has an infinite evaluation. Therefore, if $t'\gamma$ does not have an infinite evaluation, we reach states of the form $Q\gamma\delta' \mid S'\gamma$ for answer substitutions δ' and states S'. If all $Q\gamma\delta'$ did not have an infinite evaluation, this would contradict our assumption that $t'\gamma, Q\gamma$ has an infinite evaluation. Thus, there must be a state $Q\gamma\delta'$ that has an infinite evaluation. We now show that there is a concretization γ' w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\delta)$ such that $\gamma\delta' = \delta\gamma'$ for all answer substitutions δ' corresponding to a successful evaluation of $t\gamma$. Then, in particular, we have an infinite evaluation from $Q\delta\gamma' \in \mathcal{CON}(Q\delta; (\mathcal{G}', \mathcal{F}', \mathcal{U}\delta))$. There are three subcases.

First, if $\mathcal{V}(t') \subseteq \mathcal{G} \cup \mathcal{F}$ we have $t'\gamma \in \mathit{PrologTerms}(\Sigma, \mathcal{F})$ as γ is a concretization and, therefore, for all $a \in \mathcal{G}(t')$, $a\gamma \in \mathit{GroundTerms}(\Sigma)$. Thus, we have $\mathit{Dom}(\delta') \subseteq \mathcal{F}(t'\gamma)$. From $\delta = \alpha_{\mathcal{F}(t')}$ we know that for all $x \in \mathcal{F}(t'\gamma) = \mathcal{F}(t')$, $x\delta \in \mathcal{A}$ is a fresh variable. We define $\gamma'(x\delta) = x\delta'$ for $x \in \mathcal{F}(t')$ and $\gamma'(x) = \gamma(x)$ otherwise. Then, obviously, $\gamma\delta' = \delta\gamma'$ 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. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\delta)$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $\mathit{Range}(\gamma'|_{\mathcal{G}'}) \subseteq \mathit{GroundTerms}(\Sigma)$, $\mathcal{F}'(\mathit{Range}(\gamma')) = \varnothing$, and $\bigwedge_{(s,s')\in\mathcal{U}\delta} s\gamma' \not\sim s'\gamma'$.

All these properties except for $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$ are shown in (Schneider-Kamp 2008).

We perform a case analysis based on the partition $\mathcal{G}' = \mathcal{G} \uplus (ApproxGnd(t', \delta) \setminus \mathcal{G})$. For $a \in \mathcal{G}$ we have defined $a\gamma' = a\gamma$ and thus $a\gamma' \in GroundTerms(\Sigma)$. For $a \in ApproxGnd(t', \delta) \setminus \mathcal{G}$ by definition of $ApproxGnd$ and equality of $\gamma\delta'$ and $\delta\gamma'$ we know that $a\gamma' \in GroundTerms(\Sigma)$.

Second, if $\mathcal{A}(t') \subseteq \mathcal{G}$, but $\mathcal{N}(t') \not\subseteq \mathcal{F}$, the answer substitution δ' can instantiate non-abstract variables in t' which might occur in the terms represented by the abstract variables in Q. However, δ' cannot instantiate non-abstract variables not occurring in t'. We define γ' in such a way that $\gamma\delta' = \delta\gamma'$ and $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)}$. This is always possible because $Dom(\delta') \cap (\mathcal{N} \setminus \mathcal{N}(t)) = \varnothing$ and all variables in $Range(\delta)$ are fresh. Then, clearly, $Q\gamma\delta' = Q\delta\gamma$. We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\delta)$. As we only need to define γ' for abstract variables, clearly $\gamma'|_{\mathcal{A}} = \gamma'$. From $\mathcal{A}(Range(\delta')) = \varnothing$ and $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$ we know that $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$. We perform a case analysis based on the partition $\mathcal{G}' = \mathcal{G} \uplus (ApproxGnd(t', \delta) \setminus \mathcal{G})$. For $a \in \mathcal{G}$ we have effectively defined $a\gamma' = a\gamma$ and thus $a\gamma' \in GroundTerms(\Sigma)$. For $a \in ApproxGnd(t', \delta) \setminus \mathcal{G}$ by definition of $ApproxGnd$ and equality of $\gamma\delta'$ and $\delta\gamma'$ we again know that $a\gamma' \in GroundTerms(\Sigma)$. Furthermore, note that w.l.o.g. $\mathcal{F}(Range(\delta')) \subseteq \mathcal{F}(t')$ and $\mathcal{F}(Range(\gamma)) = \varnothing$. Thus, $\mathcal{F}(Range(\gamma')) \subseteq \mathcal{F}(t')$ and, consequently, $\mathcal{F}'(Range(\gamma')) = \varnothing$. For all $(s, s') \in \mathcal{U}$ we have $s\gamma \not\sim s'\gamma$ and, consequently $s\gamma\delta' \not\sim s'\gamma\delta'$. But from $s\gamma\delta' = s\delta\gamma'$ and $s'\gamma\delta' = s'\delta\gamma'$ we get $s\delta\gamma' \not\sim s'\delta\gamma'$. Thus, for all $(s'', s''') \in \mathcal{U}\delta$, we have $s\gamma' \not\sim s'\gamma'$.

Third, if $\mathcal{V}(t') \not\subseteq \mathcal{G} \cup \mathcal{F}$, the answer substitution δ' can potentially instantiate any non-ground term in $Q\gamma$ except for variables from $\mathcal{F}(Q) \setminus \mathcal{F}(t')$. We define γ' in such a way that $\gamma\delta' = \delta\gamma'$ and $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)}$. This is always possible because $Dom(\delta') \cap (\mathcal{F} \setminus \mathcal{F}(t')) = \varnothing$ and all variables in $Range(\delta)$ are fresh. Then, clearly, $Q\gamma\delta' = Q\delta\gamma'$. We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\delta)$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, $\mathcal{F}'(Range(\gamma')) = \varnothing$, and $\bigwedge_{(s,s')\in\mathcal{U}\delta} s\gamma' \not\sim s'\gamma'$.

All these properties except for $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$ are shown in (Schneider-Kamp 2008).

We perform a case analysis based on the partition $\mathcal{G}' = \mathcal{G} \uplus (ApproxGnd(t', \delta) \setminus \mathcal{G})$. For $a \in \mathcal{G}$ we have effectively defined $a\gamma' = a\gamma$ and thus $a\gamma' \in GroundTerms(\Sigma)$. For $a \in ApproxGnd(t', \delta) \setminus \mathcal{G}$ by definition of $ApproxGnd$ and equality of $\gamma\delta'$ and $\delta\gamma'$ we again know that $a\gamma' \in GroundTerms(\Sigma)$. □

Lemma A 11 (*Soundness of* INST *(Ströder 2010)*)
The rule INSTANCE is sound. Additionally, for every concretization γ w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$ there is a concretization γ' w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}')$ such that $S\gamma = S''\gamma'\mu|_{\mathcal{N}}$.

Proof
Assume we have an infinite evaluation starting from $S\gamma \in \mathcal{CON}(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$. We show that there is a substitution γ' such that $S'\gamma' \in \mathcal{CON}(S'; (\mathcal{G}', \mathcal{F}', \mathcal{U}'))$ and $S'\gamma'$ has an infinite evaluation.

For this purpose, we first show that $S''\gamma' \in \mathcal{CON}(S''; (\mathcal{G}', \mathcal{F}', \mathcal{U}'))$ has an infinite evaluation.

Following the proof in (Schneider-Kamp 2008), there must be a μ^{-1} such that $\mu|_{\mathcal{N}}\mu^{-1} = \mu^{-1}\mu|_{\mathcal{N}} = id$ as $\mu|_{\mathcal{N}}$ is a variable renaming. 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 evaluation. 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. $(\mathcal{G}', \mathcal{F}', \mathcal{U}')$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $\mathit{Range}(\gamma'|_{\mathcal{G}'}) \subseteq \mathit{GroundTerms}(\Sigma)$, $\mathcal{F}'(\mathit{Range}(\gamma')) = \varnothing$, and $\bigwedge_{(t,t')\in\mathcal{U}} t\gamma' \not\approx t'\gamma'$.

All these properties except for $\mathit{Range}(\gamma'|_{\mathcal{G}'}) \subseteq \mathit{GroundTerms}(\Sigma)$ are shown in (Schneider-Kamp 2008).

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

Since S'' is a scope variant of S' and γ' replaces only abstract variables, $S''\gamma'$ is also a scope variant of $S'\gamma'$. As $S''\gamma' \in \mathcal{CON}(S''; (\mathcal{G}', \mathcal{F}', \mathcal{U}'))$ has an infinite evaluation, we obtain by Lemma A 8 that $S'\gamma' \in \mathcal{CON}(S'; (\mathcal{G}', \mathcal{F}', \mathcal{U}'))$ has an infinite evaluation, too. □

Appendix B Extended Definitions

We first restrict the INST rule again such that it does not lose any ground information as in the paper. So we only consider derivation graphs where for every node $n \in \mathit{Inst}(G)$ with $n = S_{inst}; \mathcal{G}_{inst}, \mathcal{F}_{inst}, \mathcal{U}_{inst}$, $\mathit{Succ}_1(s) = S_{of}; \mathcal{G}_{of}, \mathcal{F}_{of}, \mathcal{U}_{of}$, and $S_{of}\mu = S_{inst}$, we have for each $x \in \mathit{Dom}(\mu)$ that if $\mathcal{V}(x\mu) \subseteq \mathcal{G}_{inst}$ holds, then $x \in \mathcal{G}_{of}$.

Moreover, we extend connections paths to the presence of PARALLEL nodes.

Definition B 1 (*Connection Path*)
A path $\pi = s_1 \dots s_k$ is a *connection path* w.r.t. G if, and only if, $k > 1$ and the following conditions are satisfied:

- $s_1 \in \{\mathit{root}(G)\} \cup \mathit{Succ}_1(\mathit{Inst}(G) \cup \mathit{Split}(G) \cup \mathit{Parallel}(G)) \cup \mathit{Succ}_2(\mathit{Split}(G) \cup \mathit{Parallel}(G))$
- $n_k \in \mathit{Inst}(G) \cup \mathit{Suc}(G) \cup \mathit{Split}(G) \cup \mathit{Parallel}(G) \cup \mathit{Succ}_1(\mathit{Inst}(G))$
- for all $1 \leq j < k$, $s_j \notin \mathit{Inst}(G) \cup \mathit{Split}(G) \cup \mathit{Parallel}(G)$
- for all $1 < j < k$, $s_j \notin \mathit{Succ}_1(\mathit{Inst}(G))$

Finally, we adapt the definition of the synthesized TRS.

Definition B 2 (*TRS from Derivation Graph*)
The TRS for a derivation graph G is defined as $\mathcal{R}(G) = \mathcal{C}(G) \cup \mathcal{S}(G) \cup \mathcal{P}(G)$ with
$\mathcal{C}(G) = \bigcup_{\pi \text{ connection path w.r.t. } G} \mathit{ConnectionRules}(\pi)$,
$\mathcal{S}(G) = \bigcup_{n \in \mathit{Split}(G)} \mathit{SplitRules}(n)$ and
$\mathcal{P}(G) = \bigcup_{n \in \mathit{Parallel}(G)} \mathit{ParallelRules}(n)$.

For a path $\pi = s_1 \dots s_k$, we define $\mathit{ConnectionRules}(\pi)$ as follows. If $s_k \in \mathit{Suc}(G)$, then $\mathit{ConnectionRules}(\pi) = \{\mathit{ren}^{in}(s_1)\sigma_{\pi,0} \to \mathit{ren}^{out}(s_1, \mathit{skip}(\pi, 1))\sigma_{\pi,0}\}$. Otherwise, we have $\mathit{ConnectionRules}(\pi) = \{$

$$\begin{array}{c} \mathit{ren}^{in}(s_1)\sigma_{\pi,0} \\ \to \\ \mathsf{u}_{s_1,s_k}(\mathit{ren}^{in}(s_k), \mathcal{V}(\mathit{ren}^{in}(s_1)))\sigma_{\pi,0} \end{array}$$

$\} \cup \bigcup_{j \in \{1,\dots,g\} \wedge \text{ the } j\text{-th goal in } s_k \text{ is no scope marker}} \{$

$$\begin{array}{c} \mathsf{u}_{s_1,s_k}(\mathit{ren}^{out}(s_k, j), \mathcal{V}(\mathit{ren}^{in}(s_1)))\sigma_{\pi,j-1} \\ \to \\ \mathit{ren}^{out}(s_1, \mathit{skip}(\pi, j))\sigma_{\pi,j-1} \end{array}$$

$\}$ where s_k has g goals.
Here, u_{s_1,s_k} is a fresh function symbol and the functions ren^{in} and ren^{out} are defined as follows:

$$ren^{in}(s) = \begin{cases} ren^{in}(Succ_1(s))\mu & \text{if } s \in \mathit{Inst}(G) \text{ where} \\ & \mu \text{ is the substitution associated with } s \\ f_s^{\text{in}}(\mathcal{G}^{in}(s)) & \text{otherwise, where } f_s^{\text{in}} \text{ is a fresh function} \\ & \text{symbol and } \mathcal{G}^{in}(S;(\mathcal{G},\mathcal{F},\mathcal{U})) = \mathcal{G} \cap \mathcal{V}(S). \end{cases}$$

$$ren^{out}(s,i) = \begin{cases} ren^{out}(Succ_1(s),i)\mu & \text{if } s \in \mathit{Inst}(G) \text{ where} \\ & \mu \text{ is the substitution associated with } s \\ f_{s,i}^{\text{out}}(\mathcal{G}^{out}(s,i)) & \text{otherwise, where } f_{s,i}^{\text{out}} \text{ is a fresh function} \\ & \text{symbol,} \\ & t_1,\ldots,t_m \text{ is the } i\text{-th goal in } s, \text{ and} \\ & \mathcal{G}^{out}(S;(\mathcal{G},\mathcal{F},\mathcal{U}),i) \\ & = (\mathcal{G} \cap \mathcal{V}(S)) \cup \\ & \mathit{ApproxG}([t_1,\ldots,t_m], \mathcal{G} \cap \\ & \mathcal{V}(S)). \end{cases}$$

The function symbols f_s^{in} and $f_{s,i}^{\text{out}}$ for all $i \in \mathbb{N}$ must be different from each other. Moreover, *ApproxG* approximates the abstract variables that have to be instantiated by ground terms using a given groundness analysis $\mathit{Ground}_{\mathcal{P}} : \Sigma \times 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ which given a predicate p and a set of ground argument positions computes the set of ground argument positions after a successful computation using the clauses from $\mathcal{P}$:

$\mathit{ApproxG}([],\mathcal{G}) = \varnothing$

$\mathit{ApproxG}([t|L],\mathcal{G}) = \mathit{NextG}(t,\mathcal{G}) \cup \mathit{ApproxG}(L, \mathcal{G} \cup \mathit{NextG}(t,\mathcal{G}))$

where

$\mathit{NextG}(t,\mathcal{G}) = \{\mathcal{V}(t_i) \mid t = p(t_1,\ldots,t_n), i \in \mathit{Ground}_{\mathit{Slice}_{\mathcal{P}}(t)}(\mathsf{p}, \{i \mid \mathcal{V}(t_i) \subseteq \mathcal{G}\})\}$

For a node $s \in \mathit{Split}(G)$, we define $\mathit{SplitRules}(s) = \{$

$$ren^{in}(s)\delta \to \mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s))\delta, \mathcal{V}(ren^{in}(s))\delta)$$

$$\mathsf{u}_{s,Succ_1(s)}(ren^{out}(Succ_1(s),1)\delta, \mathcal{V}(ren^{in}(s))\delta) \to \mathsf{u}_{Succ_1(s),Succ_2(s)}(ren^{in}(Succ_2(s)), (\mathcal{V}(ren^{in}(s)) \cup \mathcal{V}(ren^{out}(Succ_1(s),1)))\delta)$$

$$\mathsf{u}_{Succ_1(s),Succ_2(s)}(ren^{out}(Succ_2(s),1), (\mathcal{V}(ren^{in}(s)) \cup \mathcal{V}(ren^{out}(Succ_1(s),1)))\delta) \to ren^{out}(s,1)\delta$$

$\}$.

Here, $\mathsf{u}_{s,Succ_1(s)}$ and $\mathsf{u}_{Succ_1(s),Succ_2(s)}$ are fresh function symbols and δ is the substitution associated with s.

For a node $s \in \mathit{Parallel}(G)$, we define $\mathit{ParallelRules}(s) = \{$

$$ren^{in}(s) \to$$

$$\mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s)), \mathcal{V}(ren^{in}(s)))$$

$\} \cup \bigcup_{j\in\{1,\ldots,g_1\}\wedge \text{ the } j\text{-th goal in } Succ_1(s) \text{ is no scope marker}} \{$

$$\begin{array}{c} \mathsf{u}_{s,Succ_1(s)}(ren^{out}(Succ_1(s), j), \mathcal{V}(ren^{in}(s))) \\ \to \\ ren^{out}(s, j) \end{array}$$

$\} \cup \{$

$$\begin{array}{c} ren^{in}(s) \\ \to \\ \mathsf{u}_{s,Succ_2(s)}(ren^{in}(Succ_2(s)), \mathcal{V}(ren^{in}(s))) \end{array}$$

$\} \cup \bigcup_{i\in\{1,\ldots,g_2\}\wedge \text{ the } i\text{-th goal in } Succ_2(s) \text{ is no scope marker}} \{$

$$\begin{array}{c} \mathsf{u}_{s,Succ_2(s)}(ren^{out}(Succ_2(s), i), \mathcal{V}(ren^{in}(s))) \\ \to \\ ren^{out}(s, g_1 + i) \end{array}$$

$\}$ where $Succ_1(s)$ has g_1 goals and $Succ_2(s)$ has g_2 goals.
Again, $\mathsf{u}_{s,Succ_1(s)}$ and $\mathsf{u}_{s,Succ_2(s)}$ are fresh function symbols.

Furthermore, $skip(\pi, i)$ and $\sigma_{\pi,i}$ are defined as follows:

$$skip(s_1 \ldots s_j, i) =$$

$$\begin{cases} i & \text{if } j = 1 \\ skip(s_1 \ldots s_{j-1}, i+1) & \text{if } (s_{j-1} \in NoUnifyCase(G) \cup \\ & UnequalsCase(G), s_j = Succ_2(s_{j-1})) \\ & \text{or } s_{j-1} \in NoUnifyFail(G) \\ skip(s_1 \ldots s_{j-1}, & \text{if } (s_{j-1} \in BacktrackSecond(G), \\ i + change(s_{j-1}, s_j)) & s_j = Succ_2(s_{j-1})) \\ & \text{or } (s_{j-1} \in Backtracking(G)) \\ & \text{or } (s_{j-1} \in VarCase(G) \text{ where } s_{j-1} \text{ has } k \\ & \text{children and } s_j = Succ(k, s_{j-1})) \\ & \text{or } (s_{j-1} \in Cut(G) \text{ and } i > 1) \\ skip(s_1 \ldots s_{j-1}, & \text{if } s_{j-1} \in Introducing(G) \\ reduce(s_{j-1}, s_j, i-1) + 1) & \\ skip(s_1 \ldots s_{j-1}, i) & \text{otherwise} \end{cases}$$

$$\sigma_{s_1 \ldots s_j, i} =$$

$$\begin{cases}
id & \text{if } j = 1 \\
\sigma_{s_1 \ldots s_{j-1}, i} AnsSub(s_{j-1}) & \text{if } (s_{j-1} \in Split(G),\ s_j = Succ_2(s_{j-1}) \\
 & \text{or } s_{j-1} \in EqualsCase(G), s_j = Succ_1(s_{j-i})) \\
 & \text{or } (s_{j-1} \in Eval(G) \cup OnlyEval(G) \cup \\
 & UnifyCase(G) \cup UnifySuccess(G), \\
 & s_j = Succ_1(s_{j-1}), i = 0) \\
\sigma_{s_1 \ldots s_{j-1}, i} BackSub(s_{j-1}) & \text{if } s_{j-1} \in Eval(G) \cup OnlyEval(G) \cup \\
 & UnifyCase(G) \cup UnifySuccess(G), \\
 & s_j = Succ_1(s_{j-1}), i > 0 \\
\sigma_{s_1 \ldots s_{j-1}, i+1} BackSub(s_{j-1}) & \text{if } (s_{j-1} \in NoUnifyCase(G) \cup \\
 & UnequalsCase(G), s_j = Succ_2(s_{j-1})) \\
 & \text{or } s_{j-1} \in NoUnifyFail(G) \\
\sigma_{s_1 \ldots s_{j-1}, i} \sigma_{c-1} & \text{if } s_{j-1} \in VarCase(G) \text{ where } s_{j-1} \text{ has more} \\
 & \text{than } c \text{ children}, s_j = Succ(c, s_{j-1}) \text{ and} \\
 & \sigma_{c-1} \text{ is the substitution used for } s_j \\
\sigma_{s_1 \ldots s_{j-1}, i + change(s_{j-1}, s_j)} & \text{if } (s_{j-1} \in BacktrackSecond(G), \\
 & s_j = Succ_2(s_{j-1})) \\
 & \text{or } (s_{j-1} \in Backtracking(G)) \\
 & \text{or } (s_{j-1} \in VarCase(G) \text{ where } s_{j-1} \text{ has } k \\
 & \text{children and } s_j = Succ(k, s_{j-1})) \\
 & \text{or } (s_{j-1} \in Cut(G) \text{ and } i > 0) \\
\sigma_{s_1 \ldots s_{j-1}, reduce(s_{j-1}, s_j, i)} & \text{if } s_{j-1} \in Introducing(G) \\
\sigma_{s_1 \ldots s_{j-1}, i} & \text{otherwise}
\end{cases}$$

Here, $AnsSub : V \to Subst(\Sigma, \mathcal{V})$ and $BackSub : V \to Subst(\Sigma, \mathcal{V})$ are defined by:

$$AnsSub(s) = \begin{cases} \sigma & \text{if } s \in EqualsCase(G) \text{ where } \sigma \text{ is the substitution} \\ & \text{used for } Succ_1(s) \\ \sigma' & \text{if } s \in Eval(G) \cup OnlyEval(G) \cup UnifyCase(G) \\ & \cup UnifySuccess(G) \text{ where } \sigma' \text{ is the substitution} \\ & \text{used for } Succ_1(s) \\ \delta & \text{if } s \in Split(G) \text{ where } \delta \text{ is the substitution} \\ & \text{used for } Succ_2(s) \\ id & \text{otherwise} \end{cases}$$

$$BackSub(s) = \begin{cases} \sigma|_{\mathcal{G}} & \text{if } s \in Eval(G) \cup NoUnifyFail(G) \cup OnlyEval(G) \\ & \cup UnifyCase(G) \cup UnifySuccess(G) \text{ where} \\ & \sigma|_{\mathcal{G}} \text{ is the substitution used for } Succ_1(s) \\ \sigma|_{\mathcal{G}} & \text{if } s \in NoUnifyCase(G) \text{ where } \sigma|_{\mathcal{G}} \text{ is the} \\ & \text{substitution used for } Succ_2(s) \\ \sigma & \text{if } s \in UnequalsCase(G) \text{ where } \sigma \text{ is the substitution} \\ & \text{used for } Succ_2(s) \\ id & otherwise \end{cases}$$

The functions $change : V \times V \to \mathbb{N}$ and $reduce : V \times V \times \mathbb{N} \to \mathbb{N}$ are defined by:

$$change(s_1, s_2) =$$

$$\begin{cases} 1 & \text{if } (s_1 \in BacktrackSecond(G) \setminus \{\textsc{Parallel}\}, s_2 = Succ_2(s_1)) \text{ or} \\ & (s_1 \in Backtracking(G)) \text{ or } (s_1 \in VarCase(G) \text{ where} \\ & s_1 \text{ has } k \text{ children and } s_2 = Succ(k, s_1)) \text{ or} \\ & (s_1 \in Call(G) \cup Disjunction(G) \cup IfThen(G) \cup Repeat(G)) \\ 2 & \text{if } s_1 \in IfThenElse(G) \cup Not(G) \\ k & \text{if } (s_1 \in Parallel(G), s_2 = Succ_2(s_1), \\ & Succ_1(s_1) = S_1 \mid \ldots \mid S_k; KB \text{ where} \\ & S_i \in Goal(\Sigma, \mathcal{V}) \ \forall i \in \{1, \ldots, k\}) \\ & \text{or } (s_1 \in Cut(G), s_1 = !_m, Q \mid S_1 \mid \ldots \mid S_k \mid ?_m \mid S; KB \\ & \text{where } S_i \in Goal(\Sigma, \mathcal{V}) \setminus \{?_m\} \ \forall i \in \{1, \ldots, k\}) \\ & \text{or } (s_1 \in Case(G), s_1 = t, Q \mid S; KB \text{ and } |Slice_{\mathcal{P}}(t)| = k) \\ 0 & \text{otherwise} \end{cases}$$

$$reduce(s_1, s_2, i) = \max(0, i - change(s_1, s_2))$$

Finally, the set $Backtracking(G)$ is defined as the union of the sets

- $AtomicFail(G)$
- $Backtrack(G)$
- $CompoundFail(G)$
- $EqualsFail(G)$
- $Fail(G)$
- $Failure(G)$
- $NonvarFail(G)$
- $Suc(G)$
- $UnequalsFail(G)$
- $UnifyFail(G)$
- $VarFail(G)$

while the set $BacktrackSecond(G)$ is defined as the union of the sets

- $AtomicCase(G)$
- $CompoundCase(G)$
- $EqualsCase(G)$
- $Eval(G)$
- $NonvarCase(G)$
- $Parallel(G)$
- $UnifyCase(G)$

and the set $Introducing(G)$ is defined as the union of the sets

- $Call(G)$
- $Case(G)$
- $Disjunction(G)$
- $IfThen(G)$
- $IfThenElse(G)$
- $Not(G)$
- $Repeat(G)$

Appendix C Proving the Correctness of the Transformation

The following lemmata show that $\mathcal{R}(G)$ can be used to simulate (at least) all evaluations of the original Prolog program by innermost rewriting and that the length of the corresponding rewrite sequences is asymptotically over-approximating the length of the original evaluations (more precisely, a number of rewrite sequences can be used as input for a function and the function's value over-approximates the length of the corresponding original evaluation).

Before we start to state the lemmata, we introduce the notions of a state prefix and extension, respectively, which will be used in the following proofs.

Definition C 1 (*State Prefix, State Extension*)
Let S be a state with $S = S_1 \mid \cdots \mid S_k$ where $\forall i \in \{1, \ldots, k\} : S_i \in \mathit{Goal}(\Sigma, \mathcal{V})$. 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 $i \in \{1, \ldots, k\}$:

- $S_i \in \mathbb{N} \implies f(S_i) = S'_i$
- $S_i = Q \implies S'_i = Q', Q''$ for some list of terms Q'' where $Q' = Q\xi$
- $S_i = (Q)^n_m \implies S'_i = (Q', Q'')^n_{f(m)}$ for some list of terms Q'' where $Q' = Q\xi$

Here, we define $\xi = [!_i/!_{f(i)} \forall i \in \mathbb{N}]$.

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

Example C 2
Consider the state $S = t_1, t_2 \mid (t_3)^i_m$. The state t_1 is a state prefix of S while the state $t_1, t_2 \mid (t_3)^i_m \mid (t_4)^{i'}_{m'}$ is a state extension of S.

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

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

Definition C 3 (*Tree Path*)
For a derivation 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 $\mathbb{N} \times V \times \mathbb{N}$ a *tree path* w.r.t. G iff the following conditions are satisfied for all $i, j \in \mathbb{N}$:

- $p_0 \notin \{n_0, n_1, n_2, \ldots\}$
- $n_i = n_j \implies i = j$
- $i \neq 0 \implies p_i \in \{n_0, n_1, n_2, \ldots\}$
- $n_i = p_j \implies (v_i, v_j) \in E$

- $p_i < n_i$
- 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 $r \in \{1, \ldots, m_i\}$
- if (n_i, v_i, p_i) has more than one successor, then we have $v_i \in \mathit{Split}(G) \cup \mathit{Parallel}(G)$
- if (n_i, v_i, p_i) with $v_i \in \mathit{Split}(G) \cup \mathit{Parallel}(G)$ has only one successor (n_j, v_j, n_i), we have $v_j = \mathit{Succ}_1(v_i)$

Here, we call (n_j, v_j, p_j) a successor of (n_i, v_i, p_i) iff $p_j = n_i$. We call (n_i, v_i, p_i) a leaf of π iff it has no successor. We call (n_0, v_0, p_0) the root of π. 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 $r \in \{1, \ldots, m_i\}$. We call a tree path π' a subtree of a tree path π iff the root of π' occurs in π and all (n_i, v_i, p_i) in π, where the root of π' is an ancestor, also occur in π'. We call a subtree π' of π direct iff the root of π' is a successor of π's root. Moreover, for each tree path π we define $\mathit{numOfSuccesses}(\pi)$ as follows:

$\mathit{numOfSuccesses}((n_0, v_0, p_0), \ldots) =$

$$\begin{cases} 1 + \Sigma_{\pi' \text{ is a direct subtree of } \pi} \mathit{numOfSuccesses}(\pi') & \text{if } v_0 \in \mathit{Suc}(G) \\ \mathit{numOfSuccesses}(\pi') \cdot \mathit{numOfSuccesses}(\pi'') & \text{if } v_0 \in \mathit{Split}(G), \pi \text{ has two direct subtrees } \pi' \text{ and } \pi'' \\ 0 & \text{if } v_0 \in \mathit{Split}(G), \pi \text{ has only one direct subtree} \\ \Sigma_{\pi' \text{ is a direct subtree of } \pi} \mathit{numOfSuccesses}(\pi') & \text{otherwise} \end{cases}$$

Finally, for each tree path π we define $\mathit{size}(\pi)$ as follows:

$\mathit{size}((n_0, v_0, p_0), \ldots) =$

$$\begin{cases} 1 + \mathit{size}(\pi') + \mathit{numOfSuccesses}(\pi') \cdot \mathit{size}(\pi'') & \text{if } v_0 \in \mathit{Split}(G), \pi \text{ has two direct subtrees } \pi' \text{ and } \pi'' \text{ where the root of } \pi' \text{ is } (n_j, v_j, n_0) \text{ and } v_j = \mathit{Succ}_1(v_0) \\ 1 + \Sigma_{\pi' \text{ is a direct subtree of } \pi} \mathit{size}(\pi') & \text{otherwise} \end{cases}$$

The following lemma shows how a concrete evaluation is simulated by a derivation graph, i.e., how a tree path through the graph is constructed for a concrete evaluation.

Lemma C.4 (Tree Paths for Evaluation in Derivation Graph)
Let $\sharp_a(S, S')$ denote the number of answer substitutions which are added between two concrete states S and S'. Let $S_{start}\gamma_{start} \in \mathcal{CON}(S_{start}; KB_{start})$ with $S_{start}; KB_{start} = s \in G$ for a derivation graph $G = (V, E)$ and there is a prefix of an evaluation with ℓ steps from $S_{start}\gamma_{start}$ to a state S^c_{end}. Then there are two finite tree paths $\pi_a = (1, v^a_1, p^a_1), \ldots, (k_a, v^a_{k_a}, p^a_{k_a})$ and $\pi_b = (1, v^b_1, p^b_1), \ldots, (k_b, v^b_{k_b}, p^b_{k_b})$ w.r.t. G with the following properties:

- $v_1^a = v_1^b = s$
- For all $i \in \{1, \ldots, k_a\}$ there are concretizations γ_i and variable renamings ρ_i on $\mathcal{N}$ such that the evaluation reaches a state extension of $S_i\gamma_i\rho_i$ where $v_i^a = S_i; KB_i$ and $S_i\gamma_i \in \mathcal{CON}(S_i; KB_i)$.
- For all $i \in \{1, \ldots, k_b\}$ there are concretizations γ_i and variable renamings ρ_i on $\mathcal{N}$ such that the evaluation reaches a state extension of $S_i\gamma_i\rho_i$ where $v_i^b = S_i; KB_i$ and $S_i\gamma_i \in \mathcal{CON}(S_i; KB_i)$.
- $\sharp_a(S_{start}\gamma_{start}, S^c_{end}) \leq numOfSuccesses(\pi_a)$
- $\ell \in \mathcal{O}(size(\pi_b))$

Proof

We perform the proof by induction over the lexicographic combination of first the length ℓ of the evaluation and second the edge relation of G'. Here, G' is like G except that it only contains outgoing edges of Inst, 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 Inst nodes. This is in contradiction to the definition of termination graphs which disallows cycles consisting only of Inst edges.

We first show that the lemma holds for nodes $S_{start}; KB_{start}$ where one of the abstract rules Inst, Parallel, or Split has been applied. Here, whenever we have to define the concretization γ_i and the variable renaming ρ_i and if these are not specified, then $\gamma_i = \gamma_{start}$ and $\rho_i = id$.

- If we applied the Inst rule to s, we have $Succ_1(s) = S_{inst}; KB''$ with $S_{start} = S'_{inst}\mu$ where S'_{inst} is a scope variant of S_{inst}. By Lemma A 11 we know that there is a concretization γ'' such that $S_{inst}\gamma'' \in \mathcal{CON}(S_{inst}; KB'')$ and $S_{start}\gamma_{start} = S'_{inst}\gamma''\mu|_{\mathcal{N}}$. As $\mu|_{\mathcal{N}}$ is a variable renaming and S'_{inst} is a scope variant of S_{inst}, we conclude that the evaluation from $S_{start}\gamma_{start}$ to a state extension of S^c_{end} can be completely simulated by a corresponding evaluation from $S_{inst}\gamma''$ to a state extension of a state $S^{c'}_{end}$ 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 evaluation from $S_{start}\gamma_{start}$ to a state extension of S^c_{end} then there also is an i-th state S'_i in the evaluation from $S_{inst}\gamma''$ to a state extension of $S^{c'}_{end}$ and $S'_i\mu|_{\mathcal{N}} = S_i$. Hence, we can use the induction hypothesis for the latter evaluation to obtain two tree paths π'_a and π'_b, each with root $S_{inst}; KB''$. To obtain π_a and π_b from π'_a and π'_b, we first modify all variable renamings by additionally adding $\mu|_{\mathcal{N}}$ $(\rho_i = \rho'_i\mu|_{\mathcal{N}})$. Then we add the node $S_{start}; KB_{start}$ as new root and start the paths with the edge from $S_{start}; KB_{start}$ to S_{inst}, KB''. Obviously, we did not change the number of answer substitutions and have $\sharp_a(S_{start}\gamma_{start}, S^c_{end}) = \sharp_a(S_{inst}\gamma'', S^{c'}_{end}) \leq numOfSuccesses(\pi'_a) = numOfSuccesses(\pi_a)$ and $\ell \in \mathcal{O}(size(\pi'_b)) = \mathcal{O}(size(\pi'_b) + 1) = \mathcal{O}(size(\pi_b))$.
- If we applied the Parallel rule to s, we reach two states $S_1; KB_{start}$ and $S_2; KB_{start}$ where $S_{start} = S_1 \mid S_2$. There are two cases depending on whether

the evaluation contains the complete evaluation of $S_1\gamma_{start}$, i.e., for each evaluation step from a state S to a state S' in the evaluation of $S_1\gamma_{start}$, there are two successive states S'' and S''' in the evaluation of $S_{start}\gamma_{start}$ such that S'' is a state extension of S, S''' is a state extension of S', and the evaluation step between S'' and S''' is performed with the same inference rule as for S and S'. If the evaluation contains such a sub-evaluation, we can apply the induction hypothesis to obtain two tree paths π'_a and π'_b for the evaluation of $S_1\gamma_{start}$, since either the evaluation of $S_{start}\gamma_{start}$ is longer or it has the same length while we have one PARALLEL edge less to consider. If the evaluation of $S_{start}\gamma_{start}$ is not longer, then we obtain the desired tree paths π_a and π_b by inserting the step from s to $Succ_1(s)$ before π'_a and π'_b, respectively. Clearly, we have $\sharp_a(S_{start}\gamma_{start}, S^c_{end}) = \sharp_a(S_1\gamma_{start}, S^{c'}_{end}) \leq numOfSuccesses(\pi'_a) = numOfSuccesses(\pi_a)$ and $\ell \in \mathcal{O}(size(\pi'_b)) = \mathcal{O}(size(\pi'_b) + 1) = \mathcal{O}(size(\pi_b))$. If the evaluation of $S_{start}\gamma_{start}$ is longer than the one of $S_1\gamma_{start}$, then we know by Lemma A 9 that $S_2\gamma_{start}$ is reached from $S_{start}\gamma_{start}$ by the evaluation or that a state $?_m$ for some $m \in \mathbb{N}$ is reached directly after completing the evaluation of $S_1\gamma_{start}$. In the latter case, the evaluation takes exactly one more step to reach $S^c_{end} = \varepsilon$ after completing the evaluation of $S_1\gamma_{start}$. Again, we obtain the desired tree paths π_a and π_b by inserting the step from s to $Succ_1(s)$ before π'_a and π'_b, respectively. Then we have $\sharp_a(S_{start}\gamma_{start}, S^c_{end}) = \sharp_a(S_1\gamma_{start}, S^{c'}_{end}) \leq numOfSuccesses(\pi'_a) = numOfSuccesses(\pi_a)$ and $\ell = \ell - 1 + 1 \in \mathcal{O}(size(\pi'_b) + 1) = \mathcal{O}(size(\pi_b))$. In the former case, we can apply the induction hypothesis to $Succ_2(s)$ to obtain two further tree paths π''_a and π''_b. The desired tree paths π_a and π_b are then built by having s as the root with two successors: π'_a or π'_b is the left subtree and π''_a or π''_b is the right subtree, respectively. Let the evaluation of $S_1\gamma_{start}$ take ℓ' steps. Then the remaining evaluation from $S_2\gamma_{start}$ to S^c_{end} takes $\ell - \ell'$ steps and we have $\ell = \ell - \ell' + \ell' \in \mathcal{O}(size(\pi''_b) + size(\pi'_b)) = \mathcal{O}(size(\pi''_b) + size(\pi'_b) + 1) = \mathcal{O}(size(\pi_b))$. Moreover, we have $\sharp_a(S_{start}\gamma_{start}, S^c_{end}) = \sharp_a(S_1\gamma_{start}, S^{c'}_{end}) + \sharp_a(S_2\gamma_{start}, S^{c'}_{end}) \leq numOfSuccesses(\pi'_a) + numOfSuccesses(\pi''_a) = numOfSuccesses(\pi_a)$. If the evaluation does not contain the complete evaluation of $S_1\gamma_{start}$, then the case is analogous to the case where the comlete evaluation of $S_1\gamma_{start}$ has the same length as the evaluation of $S_{start}\gamma_{start}$, because we know by Lemma A 9 that a state prefix of S^c_{end} must be reachable from $S_1\gamma_{start}$.

- If we applied the SPLIT rule to s, we know that $S_{start} = t, Q$, $Succ_1(s) = t; KB_{start}$ and $Succ_2(s) = Q\delta; KB'$. We use the induction hypothesis on $Succ_1(s)$ to obtain two tree paths π^t_a and π^t_b. There are two cases depending on whether the evaluation produces an answer substitution (in the evaluation, this will only be a candidate for an answer substitution) for $t\gamma_{start}$.

If no answer substitution is produced for $t\gamma_{start}$ during the evaluation from $S_{start}\gamma_{start}$ to S^c_{end}, then we obtain the desired tree paths by simply adding s before the root of π^t_a and π^t_b. We obtain $\sharp_a(S_{start}\gamma_{start}, S^c_{end}) = 0 \leq numOfSuccesses(\pi_a)$ and $\ell \in \mathcal{O}(size(\pi^t_b)) = \mathcal{O}(size(\pi^t_b) + 1) = \mathcal{O}(size(\pi_b))$.

If $t\gamma_{start}$ produces at least one answer substitution during the evaluation from $S_{start}\gamma_{start}$ to S^c_{end}, then we consider the two (maybe equal) answer substitutions δ_a and δ_b for $t\gamma_{start}$ such that the evaluation of $Q\gamma_{start}\delta_a$ produces as many answer substitutions as possible during the evaluation from $S_{start}\gamma_{start}$ to S^c_{end} and the eval-

uation of $Q\gamma_{start}\delta_b$ is part of the evaluation from $S_{start}\gamma_{start}$ to S^c_{end} and as long as possible. By Lemma A 10 we know that there are concretizations γ_a and γ_b such that $Q\gamma_{start}\delta_a = Q\delta\gamma_a$ and $Q\gamma_{start}\delta_b = Q\delta\gamma_b$. Thus, we can use the induction hypothesis for $Q\delta\gamma_a$ to obtain a tree path π^Q_a. The desired tree path π_a is then built by having s as its root and π^t_a as its first direct subtree and π^Q_a as its second direct subtree. We obtain

$\sharp_a(S_{start}\gamma_{start}, S^c_{end}) = \Sigma_{\delta' \text{ is an answer substitution for } t\gamma_{start} \text{ during the evaluation from } S_{start}\gamma_{start} \text{ to } S^c_{end}} \text{dur-}\sharp_a(Q\gamma_{start}\delta', S^c_{end}) \leq$ $\mathit{numOfSuccesses}(\pi^t_a) \cdot \mathit{numOfSuccesses}(\pi^Q_a) = \mathit{numOfSuccesses}(\pi_a)$. Moreover, we use the induction hypothesis for $Q\delta\gamma_b$ to obtain a tree path π^Q_b. There are two cases. If $\mathit{size}(\pi^t_b) \geq \mathit{numOfSuccesses}(\pi^t_a) \cdot \mathit{size}(\pi^Q_b)$, then the desired tree path π_b is built by having s as its root and π^t_b as its only direct subtree. Let ℓ' be the length of the evaluation of $t\gamma_{start}$ during the evaluation from $S_{start}\gamma_{start}$ to S^c_{end}. We obtain $\ell = \ell' + \Sigma_{\delta' \text{ is an answer substitution for } t\gamma_{start} \text{ during the evaluation from } S_{start}\gamma_{start} \text{ to } S^c_{end}} \text{dur-}(\ell - \ell') \in \mathcal{O}(\mathit{size}(\pi^t_b) + \mathit{numOfSuccesses}(\pi^t_a) \cdot \mathit{size}(\pi^Q_b)) = \mathcal{O}(\mathit{size}(\pi^t_b)) = \mathcal{O}(\mathit{size}(\pi^t_b) + 1) = \mathcal{O}(\mathit{size}(\pi_b))$. If $\mathit{size}(\pi^t_b) < \mathit{numOfSuccesses}(\pi^t_a) \cdot \mathit{size}(\pi^Q_b)$, then the desired tree path π_b is built by having s as its root, π^t_a as its first direct subtree and π^Q_b as its second direct subtree. We obtain $\ell = \ell' + \Sigma_{\delta' \text{ is an answer substitution for } t\gamma_{start} \text{ during the evaluation from } S_{start}\gamma_{start} \text{ to } S^c_{end}} \text{dur-}(\ell - \ell') \in \mathcal{O}(\mathit{size}(\pi^t_b) + \mathit{numOfSuccesses}(\pi^t_a) \cdot \mathit{size}(\pi^Q_b)) = \mathcal{O}(\mathit{numOfSuccesses}(\pi^t_a) \cdot \mathit{size}(\pi^Q_b)) \subseteq \mathcal{O}(1 + \mathit{size}(\pi^t_a) + \mathit{numOfSuccesses}(\pi^t_a) \cdot \mathit{size}(\pi^Q_b)) = \mathcal{O}(\mathit{size}(\pi_b))$.

For $\ell = 0$ we know that $S_{start}\gamma_{start} = S^c_{end} \in \mathcal{CON}(S_{start}; KB_{start})$. Thus, for $\gamma_0 = \gamma_{start}$ and $\rho_0 = id$ we trivially obtain $\pi_a = \pi_b = (1, s, 0)$ as the desired tree path.

For $\ell > 0$, we can assume that the lemma holds for evaluations of length at most $\ell - 1$.

We perform a case analysis over the first concrete inference rule applied in the evaluation where we can assume that the abstract inference rules Inst, Parallel, and Split were not applied to s.

- For Case we have $S_{start} = t, Q \mid S_r$ where $\mathit{root}(t) \notin \mathit{BuiltInPredicates}$ and $\mathit{Slice}_{\mathcal{P}}(t) \neq \varnothing$ and the evaluation reaches the state $(t, Q)^{i_1}_j\gamma_{start} \mid \cdots \mid (t, Q)^{i_m}_j\gamma_{start} \mid S_r\gamma_{start}$. So the only applicable abstract inference rule for s is Case.

By applying the Case rule to s, we reach the state $s' = (t, Q)^{i_1}_j \mid \cdots \mid (t, Q)^{i_m}_j \mid S_r; KB_{start}$. By the induction hypothesis we obtain the tree paths π'_a and π'_b with the properties in Lemma C 4 for s'. We obtain the desired tree paths π_a and π_b by inserting the path from s to s' before π'_a and π'_b, respectively.

- For Suc we have $S_{start} = \square \mid S_r$ and the evaluation reaches the state $S_r\gamma_{start}$. So the only applicable abstract inference rule for s is Suc.

By applying the Suc rule, we reach the state $S_r; KB_{start}$. By the induction hypothesis we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $S_r; KB_{start}$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $S_r; KB_{start}$ before π'_a and π'_b, respectively, using γ_{start} and id for $S_r; KB_{start}$.

• For FAILURE, CUT, and CUTALL, the proof is analogous to the case where the SUC rule is the first rule in the evaluation.
• For VARIABLEERROR, we have $S_{start} = \mathsf{call}(x), Q \mid S_r$ and the evaluation reaches the state ε. So the only applicable abstract inference rule for s is VARIABLEERROR. By applying the VARIABLEERROR rule, we reach the state $\varepsilon; KB_{start}$. As the evaluation has to end here, we obtain the desired tree paths $\pi_a = \pi_b = (1, S_{start}; KB_{start}, 0), (2, \varepsilon; KB_{start}, 1)$ using id and id for $\varepsilon; KB_{start}$.
• For UNDEFINEDERROR, THROW, HALT, and HALT1, the proof is analogous to the case where the VARIABLEERROR rule is the first rule in the evaluation.
• For EVAL we have $S_{start} = (t, Q)^i_j \mid S_r$ and the evaluation reaches the state $B'_i\sigma, Q\gamma_{start}\sigma \mid S_r\gamma_{start}$ as defined in the EVAL rule. From the soundness proof of BACKTRACK we know that the only applicable abstract inference rules for s are EVAL and ONLYEVAL.
If we applied the EVAL rule, we have $Succ_1(s) = B'_i\sigma', Q\sigma' \mid S_r\sigma|_{\mathcal{G}}; KB'$ as defined in EVAL. From the soundness proof of EVAL we know that there is a concretization γ'' w.r.t. KB' with $B'_i\sigma'\gamma'', Q\sigma'\gamma'' \mid S_r\sigma|_{\mathcal{G}}\gamma'' = B'_i\sigma, Q\gamma_{start}\sigma \mid S_r\gamma_{start}$. By the induction hypothesis we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $B'_i\sigma', Q\sigma' \mid S_r\sigma|_{\mathcal{G}}; KB'$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $B'_i\sigma', Q\sigma' \mid S_r\sigma|_{\mathcal{G}}; KB'$ before π'_a and π'_b, respectively, using γ'' and id for $B'_i\sigma', Q\sigma' \mid S_r\sigma|_{\mathcal{G}}; KB'$.
If we applied the ONLYEVAL rule, we have $Succ_1(s) = B'_i\sigma', Q\sigma' \mid S_r\sigma|_{\mathcal{G}}; KB'$ again and, hence, the same case as for EVAL.
• For BACKTRACK we have $S_{start} = (t, Q)^i_j \mid S_r$ and the evaluation reaches the state $S_r\gamma_{start}$. From the soundness proof of ONLYEVAL, we know that the only applicable abstract inference rules for s are EVAL and BACKTRACK.
If we applied the EVAL rule, we have $Succ_2(s) = S_r; KB'$ as defined in EVAL where we know by the soundness proof of EVAL that γ_{start} is a concretization w.r.t. KB'. By the induction hypothesis we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $S_r; KB'$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $S_r; KB'$ before π'_a and π'_b, respectively, using γ_{start} and id for $S_r; KB'$.
If we applied the BACKTRACK rule, we have $Succ_1(s) = S_r; KB'$ and, hence, the same case for $Succ_1(s)$ here as for $Succ_2(s)$ in the case of EVAL.
• For CALL we have $S_{start} = \mathsf{call}(t'), Q \mid S_r$ and the evaluation reaches the state $t''\gamma_{start}, Q\gamma_{start} \mid ?_m \mid S_r\gamma_{start}$ for $t'' = \mathit{Transformed}(t', m)$. So the only applicable abstract inference rule for s is CALL.
By applying the CALL rule, we reach the state $t'', Q \mid ?_m \mid S_r; KB_{start}$. By the induction hypothesis, we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $t'', Q \mid ?_m \mid S_r; KB_{start}$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $t'', Q \mid ?_m \mid S_r; KB_{start}$ before π'_a and π'_b, respectively, using γ_{start} and id for $t'', Q \mid ?_m \mid S_r; KB_{start}$.
• For ATOMICFAIL we have $S_{start} = \mathsf{atomic}(t'), Q \mid S_r$ where $t'\gamma_{start}$ is no constant and the evaluation reaches the state $S_r\gamma_{start}$. So the only applicable abstract inference rules for s are ATOMICFAIL and ATOMICCASE.
If we applied the ATOMICCASE rule, we have $Succ_2(s) = S_r; KB_{start}$. By the in-

duction hypothesis we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $S_r; KB_{start}$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $S_r; KB_{start}$ before π'_a and π'_b, respectively, using γ_{start} and id for $S_r; KB_{start}$.

If we applied the AtomicFail rule, we have $Succ_1(s) = S_r; KB_{start}$ and, hence, the same case for $Succ_1(s)$ here as for $Succ_2(s)$ in the case of AtomicCase.

• For AtomicSuccess we have $S_{start} = \mathsf{atomic}(t'), Q \mid S_r$ where $t'\gamma_{start}$ is a constant and the evaluation reaches the state $Q\gamma_{start} \mid S_r\gamma_{start}$. So the only applicable abstract inference rules for s are AtomicSuccess and AtomicCase.

If we applied the AtomicCase rule, we have $Succ_1(s) = Q \mid S_r; KB'$ as defined for AtomicCase where we know by the soundness proof of AtomicCase that γ_{start} is a concretization w.r.t. KB'. By the induction hypothesis we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $Q \mid S_r; KB'$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $Q \mid S_r; KB'$ before π'_a and π'_b, respectively, using γ_{start} and id for $Q \mid S_r; KB'$.

If we applied the AtomicSuccess rule, we have $Succ_1(s) = Q \mid S_r; KB'$ again and, hence, the same case as for AtomicCase.

• For CompoundFail, EqualsFail, and NonvarFail, the proof is analogous to the case for AtomicFail.

• For CompoundSuccess, NonvarSuccess, NoUnifySuccess, and UnequalsSuccess, the proof is analogous to the case for AtomicSuccess.

• For Conjunction we have $S_{start} = ,(t_1, t_2), Q \mid S_r$ and the evaluation reaches the state $t_1, t_2, Q \mid S_r$. So the only applicable abstract inference rule for s is Conjunction.

By applying the Conjunction rule, we have $Succ_1(s) = t_1, t_2, Q \mid S_r; KB_{start}$. By the induction hypothesis, we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $t_1, t_2, Q \mid S_r; KB_{start}$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $t_1, t_2, Q \mid S_r; KB_{start}$ before π'_a and π'_b, respetively, using γ_{start} and id for $t_1, t_2, Q \mid S_r; KB_{start}$.

• For Disjunction, IfThen, IfThenElse, Not, Once, and Repeat, the proof is analogous to the case where the Conjunction rule is the first rule in the evaluation.

• For EqualsSuccess we have $S_{start} = ==(t_1, t_2), Q \mid S_r$ where $t_1\gamma_{start} = t_2\gamma_{start}$ and the evaluation reaches the state $Q\gamma_{start} \mid S_r\gamma_{start}$. So the only applicable abstract inference rules for s are EqualsSuccess and EqualsCase.

If we applied the EqualsCase rule, we have $Succ_1(s) = Q\sigma \mid S_r\sigma; KB'$ as defined for EqualsCase where we know by the soundness proof of EqualsCase that $\gamma_{start} = \sigma\gamma_{start}$ is a concretization w.r.t. KB'. By the induction hypothesis, we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $Q\sigma \mid S_r\sigma; KB'$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $Q\sigma \mid S_r\sigma; KB'$ before π'_a and π'_b, respetively, using γ_{start} and id for $Q\sigma \mid S_r\sigma; KB'$.

If we applied the EqualsSuccess rule, we have $t_1 = t_2$ and $Succ_1(s) = Q \mid S_r; KB_{start}$. By the induction hypothesis, we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $Q \mid S_r; KB_{start}$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $Q \mid S_r; KB_{start}$ before π'_a and π'_b, respectively, using γ_{start} and id for $Q \mid S_r; KB_{start}$.

• For FAIL we have $S_{start} = \mathsf{fail}, Q \mid S_r$. So the only applicable abstract inference rule for s is FAIL.
By applying the FAIL rule, we have $Succ_1(s) = S_r; KB_{start}$. By the induction hypothesis, we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $S_r; KB_{start}$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $S_r; KB_{start}$ before π'_a and π'_b, respectively, using γ_{start} and id for $S_r; KB_{start}$.
• For NEWLINE we have $S_{start} = \mathsf{nl}, Q \mid S_r$. So the only applicable abstract inference rule for s is NEWLINE.
By applying the NEWLINE rule, we have $Succ_1(s) = Q \mid S_r; KB_{start}$. By the induction hypothesis, we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $Q \mid S_r; KB_{start}$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $Q \mid S_r; KB_{start}$ before π'_a and π'_b, respectively.
• For NOUNIFYFAIL we have $S_{start} = \backslash{=}(t_1, t_2), Q \mid S_r$ where $t_1\gamma_{start} \sim t_2\gamma_{start}$ and the evaluation reaches the state $S_r\gamma_{start}$. From the soundness proof of NOUNIFYSUCCESS we know that the only applicable abstract inference rules for s are NOUNIFYCASE and NOUNIFYFAIL.
If we applied the NOUNIFYCASE rule, we have $Succ_2(s) = S_r\sigma|_{\mathcal{G}}; KB'$ as defined in NOUNIFYCASE. From the soundness proof of NOUNIFYCASE we know that $\gamma_{start} = \sigma_{\mathcal{G}}\gamma_{start}$ is a concretization w.r.t. KB'. By the induction hypothesis, we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $S_r\sigma|_{\mathcal{G}}; KB'$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $S_r\sigma|_{\mathcal{G}}; KB'$ before π'_a and π'_b, respectively, using γ_{start} and id for $S_r\sigma|_{\mathcal{G}}; KB'$.
If we applied the NOUNIFYFAIL rule, we have $Succ_1(s) = S_r\gamma_{start}|_{\mathcal{G}}; KB'$ again and, hence the same case for $Succ_1(s)$ here as for $Succ_2(s)$ in the case of NOUNIFYCASE.
• For TRUE, the proof is analogous to the case for NEWLINE.
• For UNEQUALSFAIL we have $S_{start} = \backslash{=}{=}(t_1, t_2), Q \mid S_r$ where $t_1\gamma_{start} = t_2\gamma_{start}$ and the evaluation reaches the state $S_r\gamma_{start}$. So the only applicable abstract inference rules for s are UNEQUALSFAIL and UNEQUALSCASE.
If we applied the UNEQUALSCASE rule, we have $Succ_2(s) = S_r\sigma; KB'$ as defined for UNEQUALSCASE where we know by the soundness proof of UNEQUALSCASE that $\gamma_{start} = \sigma\gamma_{start}$ is a concretization w.r.t. KB'. By the induction hypothesis, we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $S_r\sigma; KB'$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $S_r\sigma; KB'$ before π'_a and π'_b, respectively, using γ_{start} and id for $S_r\sigma; KB'$.
If we applied the UNEQUALSFAIL rule, we have $t_1 = t_2$ and $Succ_1(s) = S_r; KB_{start}$. By the induction hypothesis, we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $S_r; KB_{start}$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $S_r; KB_{start}$ before π'_a and π'_b, respectively, using γ_{start} and id for $S_r; KB_{start}$.
• For UNIFYFAIL, the proof is analogous to the case for BACKTRACK.
• For UNIFYSUCCESS, the proof is analogous to the case for EVAL.
• For VARFAIL we have $S_{start} = \mathsf{var}(t'), Q \mid S_r$ where $t'\gamma_{start}$ is no variable and the evaluation reaches the state $S_r\gamma_{start}$. So the only applicable abstract inference rules for s are VARFAIL and VARCASE.

If we applied the VarCase rule and s has j children, we have $Succ_j(s) = S_r; KB_{start}$. By the induction hypothesis we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $S_r; KB_{start}$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $S_r; KB_{start}$ before π'_a and π'_b, respectively, using γ_{start} and id for $S_r; KB_{start}$.
If we applied the VarFail rule, we have $Succ_1(s) = S_r; KB_{start}$ and, hence, the same case for $Succ_1(s)$ here as for $Succ_j(s)$ in the case of VarCase.
• For VarSuccess we have $S_{start} = \mathsf{var}(t'), Q \mid S_r$ where $t'\gamma_{start} \in \mathcal{N}$ and the evaluation reaches the state $Q\gamma_{start} \mid S_r\gamma_{start}$. So the only applicable abstract inference rules for s are VarSuccess and VarCase.
Since $t'\gamma_{start} \in \mathcal{N}$, we know that $t' \in \mathcal{A} \setminus \mathcal{G}$.
If we applied the VarCase rule and s has j children, there are two cases depending on whether $t'\gamma_{start} \in \mathcal{N}(Q) \cup \mathcal{N}(S_r) \cup \mathcal{N}(KB_{start})$.
If $t'\gamma_{start} \in \mathcal{N}(Q) \cup \mathcal{N}(S_r) \cup \mathcal{N}(KB_{start})$, then there is an index j' with $1 < j' < j$ and $Succ'_j(s) = Q\sigma_{j'+1} \mid S_r\sigma_{j'+1}; KB_{start}\sigma_{j'+1}$ where $\sigma_{j'+1} = [t'/t'\gamma_{start}]$. Thus, we have $Q\sigma_{j'+1}\gamma_{start} \mid S_r\sigma_{j'+1}\gamma_{start} = Q\gamma_{start} \mid S_r\gamma_{start}$ and γ_{start} is a concretization w.r.t. $KB_{start}\sigma_{j'+1}$. By the induction hypothesis we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $Q\sigma_{j'+1} \mid S_r\sigma_{j'+1}; KB_{start}\sigma_{j'+1}$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $Q\sigma_{j'+1} \mid S_r\sigma_{j'+1}; KB_{start}\sigma_{j'+1}$ before π'_a and π'_b, respectively.
If $t'\gamma_{start} \notin \mathcal{N}(Q) \cup \mathcal{N}(S_r) \cup \mathcal{N}(KB_{start})$, then we have $Succ_1(s) = Q\sigma_0 \mid S_r\sigma_0; KB_{start}\sigma_0$ where $\sigma_0 = [t'/x]$ and $x \in \mathcal{N}_{fresh}$. By the soundness proof of VarCase, we know that there is a concretization γ'' w.r.t. $KB_{start}\sigma_0$ and a variable renaming ρ' on $\mathcal{N}$ such that $\gamma_{start}\rho' = \gamma''$ and $Q\sigma_0\gamma_{start}\rho' \mid S_r\sigma_0\gamma_{start}\rho' = Q\sigma_0\gamma'' \mid S_r\sigma_0\gamma'' = Q\gamma'' \mid S_r\gamma''$. By the induction hypothesis, we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $Q\sigma_0 \mid S_r\sigma_0; KB_{start}\sigma_0$. Thus, using γ_{start} and ρ' for $Q\sigma_0 \mid S_r\sigma_0; KB_{start}\sigma_0$ we obtain the desired tree paths π_a and π_b by inserting the path from s to $Q\sigma_0 \mid S_r\sigma_0; KB_{start}\sigma_0$ before π'_a and π'_b, respectively.
If we applied the VarSuccess rule, then we have $Succ_1(s) = Q \mid S_r; KB_{start}$. By the induction hypothesis, we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $Q \mid S_r; KB_{start}$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $Q \mid S_r; KB_{start}$ before π'_a and π'_b, respectively.
• For Write we have $S_{start} = \mathsf{write}(t'), Q \mid S_r$. So the only applicable abstract inference rule for s is Write.
By applying the Write rule, we have $Succ_1(s) = Q \mid S_r; KB_{start}$. By the induction hypothesis, we obtain two tree paths π'_a and π'_b with the properties in Lemma C 4 for $Q \mid S_r; KB_{start}$. Thus, we obtain the desired tree paths π_a and π_b by inserting the path from s to $Q \mid S_r; KB_{start}$ before π'_a and π'_b, respectively.
• For WriteCanonical and Writeq, the proof is analogous to the case for Write.

□

Next, we show that we can use the same concretization and variable renaming as long as we do not traverse Inst nodes.

Lemma C 5 (*Single Concretization and Variable Renaming*)
Given a path $\pi = s_1 \dots s_k$ with $n_j \notin \mathit{Inst}(G)$ for all $j \in \{1, \dots, k-1\}$ and an evaluation 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 evaluation 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 $s_1 = s_k$ 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 Inst, we know that only fresh variables are introduced by these rules. We perform a case analysis over s_1 and s_2.

• If $s_1 \in \mathit{Split}(G)$ and $s_2 = \mathit{Succ}_2(s_1)$, i.e., we traverse the right child of a Split node, we have $s_1 = t, Q; KB$ and $s_2 = Q\mu; KB'$ 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 A 10 and the fact that the evaluation 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 $i \in \{1, \dots, k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2, \dots, k\}$. 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 $s_1 \in \mathit{Eval}(G)$ and $s_2 = \mathit{Succ}_1(s_1)$, i.e., we traverse the left child of an Eval node, we have $s_1 = (t, Q)^{\mathrm{c}}_m \mid S; KB$ and $s_2 = B'_c\sigma', Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ 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 $j \in \{2, \dots, k\}$ 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 the soundness proof of Eval and the fact that the evaluation 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 $(t, Q)^{\mathrm{c}}_m\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with answer substitution σ'' and $\rho_1 = \rho_2$, we obtain $\gamma_1\rho_1\sigma'' = \sigma'\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$. 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 $i \in \{1,\dots,k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2,\dots,k\}$. 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 $s_1 \in \mathit{OnlyEval}(G)$ and $s_2 = \mathit{Succ}_1(s_1)$, i.e., we traverse an OnlyEval node, we have $s_1 = (t,Q)^c_m \mid S;KB$ and $s_2 = B'_c\sigma', Q\sigma' \mid S\sigma|_{\mathcal{G}};KB'$ as defined in the OnlyEval 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 $B'_c\sigma'\gamma_2\rho_2, Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2 = B'_c\sigma'\gamma'\rho, Q\sigma'\gamma'\rho \mid S\sigma|_{\mathcal{G}}\gamma'\rho$. By the soundness proof of OnlyEval and the fact that the evaluation 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'' and $\rho_1 = \rho_2$, we obtain $\gamma_1\rho_1\sigma'' = \sigma'\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$. 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 $i \in \{1,\dots,k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2,\dots,k\}$. 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 $s_1 \in \mathit{UnifyCase}(G)$ and $s_2 = \mathit{Succ}_1(s_1)$, i.e., we traverse the left child of a UnifyCase node, we have $s_1 = {=}(t_1,t_2), Q \mid S;KB$ and $s_2 = Q\sigma' \mid S\sigma|_{\mathcal{G}};KB'$ as defined in the UnifyCase 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 $Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2 = Q\sigma'\gamma'\rho \mid S\sigma|_{\mathcal{G}}\gamma'\rho$. By the soundness proof of UnifyCase and the fact that the evaluation reaches a state extension of $Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2$ from a state extension of $({=}(t_1,t_2),Q)\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with answer substitution σ'' and $\rho_1 = \rho_2$, we obtain $\gamma_1\rho_1\sigma'' = \sigma'\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$. Since only fresh abstract variables are introduced along π, we have for all abstract variables $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)) \setminus (\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ 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 $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)) \setminus (\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ and $T\gamma = T\gamma'$ otherwise. Then we obviously have $S_i\gamma\rho = S_i\gamma_i\rho_i$ for all $i \in \{1,\dots,k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2,\dots,k\}$. 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 $s_1 \in \mathit{UnifySuccess}(G)$ and $s_2 = \mathit{Succ}_1(s_1)$, i.e., we traverse a UnifySuccess node, we have $s_1 = {=}(t_1,t_2), Q \mid S;KB$ and $s_2 = Q\sigma' \mid S\sigma|_{\mathcal{G}};KB'$ as defined in the UnifySuccess 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 $Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2 = Q\sigma'\gamma'\rho \mid S\sigma|_{\mathcal{G}}\gamma'\rho$. By the soundness proof of UnifySuccess and the fact that the evaluation reaches a state extension of $Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2$ from a state extension of $(=(t_1,t_2),Q)\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with answer substitution σ'' and $\rho_1 = \rho_2$, we obtain $\gamma_1\rho_1\sigma'' = \sigma'\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$. Since only fresh abstract variables are introduced along π, we have for all abstract variables $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)) \setminus (\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ 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 $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)) \setminus (\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ 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.

• If $s_1 \in \mathit{NoUnifyCase}(G)$ and $s_2 = \mathit{Succ}_2(s_1)$, i.e., we traverse the right child of a NoUnifyCase node, we have $s_1 = \backslash{=}(t_1,t_2),Q \mid S;KB$ and $s_2 = S\sigma|_{\mathcal{G}};KB'$ as defined in the NoUnifyCase 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 $S\sigma|_{\mathcal{G}}\gamma_2\rho_2 = S\sigma|_{\mathcal{G}}\gamma'\rho$. By the soundness proof of NoUnifyCase and the fact that the evaluation reaches a state extension of $S\sigma|_{\mathcal{G}}\gamma_2\rho_2$ from a state extension of $(\backslash{=}(t_1,t_2),Q)\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with answer substitution σ'' and $\rho_1 = \rho_2$, we obtain $\gamma_1\rho_1\sigma'' = \sigma'\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$. Since only fresh abstract variables are introduced along π, we have for all abstract variables $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)) \setminus (\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ 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 $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)) \setminus (\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ 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.

• If $s_1 \in \mathit{NoUnifyFail}(G)$ and $s_2 = \mathit{Succ}_1(s_1)$, i.e., we traverse a NoUnifyFail node, we have $s_1 = \backslash{=}(t_1,t_2),Q \mid S;KB$ and $s_2 = S\sigma|_{\mathcal{G}};KB'$ as defined in the NoUnifyFail 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 $S\sigma|_{\mathcal{G}}\gamma_2\rho_2 = S\sigma|_{\mathcal{G}}\gamma'\rho$. By the soundness proof of NoUnifyFail and the fact that the evaluation reaches a state extension of $S\sigma|_{\mathcal{G}}\gamma_2\rho_2$ from a state extension of $(\backslash{=}(t_1,t_2),Q)\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with answer substitution σ'' and $\rho_1 = \rho_2$, we obtain $\gamma_1\rho_1\sigma'' = \sigma'\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$. Since only fresh abstract variables are introduced along π, we have for all abstract variables $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)) \setminus (\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ 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 $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)) \setminus (\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ 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.

• If $s_1 \in \mathit{VarCase}(G)$ and $s_2 = \mathit{Succ}_1(s_1)$, i.e., we traverse the first child of a VarCase node where we introduce a fresh non-abstract variable, we have $s_1 = \mathsf{var}(a), Q \mid S; KB$ with $a \in \mathcal{A} \setminus \mathcal{G}$ and $s_2 = Q\sigma_0 \mid S\sigma_0; KB\sigma_0$ with $\sigma_0 = [a/x]$ and $x \in \mathcal{N}_{\mathit{fresh}}$ as defined in the VarCase rule. By the induction hypothesis we obtain a variable renaming ρ' and a concretization γ' w.r.t. KB_j for all $j \in \{2,\ldots,k\}$ such that $S_j\gamma_j\rho_j = S_j\gamma'\rho'$. In particular, we have $Q\sigma_0\gamma_2\rho_2 \mid S\sigma_0\gamma_2\rho_2 = Q\sigma_0\gamma'\rho' \mid S\sigma_0\gamma'\rho'$. By the soundness proof of VarCase and the fact that the evaluation reaches a state extension of $Q\sigma_0\gamma_2\rho_2 \mid S\sigma_0\gamma_2\rho_2$ from a state extension of $(\mathsf{var}(a), Q)\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with an empty answer substitution, we obtain a variable renaming ρ such that $\gamma_1\rho_1\rho = \sigma_0\gamma_1\rho_2 = \sigma_0\gamma_2\rho_2$ and $a'\gamma_1 = a'\gamma_2$ for $a' \neq a$. We define γ by $a'\gamma = a'\gamma'$ for $a' \neq a$ and $a\gamma = a\gamma_1$. Since x is fresh and only fresh variables are introduced along π, we have for all non-abstract variables $x' \in (\mathcal{N}(Q) \cup \mathcal{N}(S) \cup \mathcal{N}(KB)) \setminus (\mathcal{N}(Q\sigma_0) \cup \mathcal{N}(S\sigma_0) \cup \mathcal{N}(KB'))$ that $x' \notin \mathcal{N}(S_j) \cup \mathcal{N}(KB_j)$. Hence, we can define the variable renaming ρ by $x\rho = a\gamma_2$ and $x'\rho = x'\rho'$ for $x' \in (\mathcal{N}(Q) \cup \mathcal{N}(S) \cup \mathcal{N}(KB)) \setminus (\mathcal{N}(Q\sigma_0) \cup \mathcal{N}(S\sigma_0) \cup \mathcal{N}(KB'))$. 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 $j \in \{2,\ldots,k\}$. Since 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.

□

Furthermore, we show that our definition of skip values follows exactly the number of backtracked or cut state elements at the beginning of a state. For this we can already use the result of Lemma C 5 and show this only for evaluations using the same concretization and variable renaming along a path. Moreover, we can assume that the path does not end in an empty state as we cannot have any connection paths with such a path as a subpath.

Lemma C 6 (*Skip Values Correspond to Backtracking or Cutting*)
Given a path $\pi = s_1 \ldots s_k$ with $k > 1$, $s_j \notin \mathit{Inst}(G)$ for all $j \in \{1,\ldots,k-1\}$, an evaluation such that there is a variable renaming ρ and a concretization γ w.r.t. $KB_1,\ldots,KB_k$ where $s_i = S_i; KB_i$ for all $i \in \{1,\ldots,k\}$ and the evaluation goes from a state extension of $S_1\gamma\rho$ to a state extension of $S_k\gamma\rho$ by reaching state extensions of all $S_i\gamma\rho$, and $S_k \neq \varepsilon$, then $\sigma_{\pi,0} = \sigma_{s_1 s_2,d}\sigma_{s_2\ldots s_k,0}$ iff the evaluation backtracks or cuts the first d state elements of the state extension of $S_2\gamma\rho$ until it reaches the state extension of $S_k\gamma\rho$.

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

For $k = 2$ we have $\sigma_{\pi,0} = \sigma_{s_1 s_2,0} = \sigma_{s_1 s_2,0} id = \sigma_{s_1 s_2,0}\sigma_{s_2,0}$ and as the evaluation cannot backtrack or cut any state elements from the state extension of S_2 to the same state extension of S_2, the lemma holds.

For $k > 2$ we can assume the lemma holds for paths of length at most $k - 1$.

By the induction hypothesis, we obtain that $\sigma_{s_2 \ldots s_k,0} = \sigma_{s_2 s_3,d'}\sigma_{s_3 \ldots s_k,0}$ iff the evaluation backtracks or cuts d' state elements of $S_3\gamma\rho$ until it reaches the state extension of $S_k\gamma\rho$. By Def. B 2 we also know that $\sigma_{\pi,0} = \sigma_{s_1 s_2,d}\sigma_{s_2 \ldots s_k}$ for some $d \in \mathbb{N}$. We perform a case analysis over s_2 and s_3.

• If $s_2 \in \mathit{NoUnifyCase}(G) \cup \mathit{UnequalsCase}(G)$ and $s_3 = \mathit{Succ}_2(s_2)$, i.e., we traverse the right child of a NoUnifyCase or UnequalsCase node, we have $d = d' + 1$. Furthermore, the evaluation backtracks exactly one state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the evaluation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' + 1 = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds.
• If $s_2 \in \mathit{Parallel}(G)$ and $s_3 = \mathit{Succ}_2(s_2)$, i.e., we traverse the right child of a Parallel node, we have $d = d' + j$ where $\mathit{Succ}_1(s_2)$ contains j state elements. Furthermore, as the evaluation reaches a state extension of $S_3\gamma\rho$ from a state extension of $S_2\gamma\rho$, it must backtrack or cut the first j state elements of $S_2\gamma\rho$ from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the evaluation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' + j = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds.
• If $s_2 \in \mathit{Cut}(G)$ and $s_3 = \mathit{Succ}_1(s_2)$, i.e., we traverse a Cut node, there are two cases depending on whether $d' = 0$. If $d' = 0$, then we have $d = 0$ and the evaluation does not backtrack or cut any state element before the first state element of the state extension of $S_3\gamma\rho$. Since this first state element of the state extension of $S_3\gamma\rho$ corresponds to the first state element of the state extension of $S_2\gamma\rho$ the evaluation does not backtrack or cut any state element before the first state element of the state extension of $S_2\gamma\rho$ and the lemma holds. If otherwise $d' > 0$, then we have $d = d' + j$ where $s_2 = !_m, Q \mid S'_1 \mid \ldots \mid S'_j \mid ?_m \mid S'$ and $S'_i \in \mathit{Goal}(\Sigma, \mathcal{V}) \setminus \{?_m\}$ $\forall i \in \{1, \ldots, j\}$. Furthermore, the evaluation cuts exactly j state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. Since $d' > 0$, these state elements must belong to the first state elements of the state extension of $S_2\gamma\rho$ which are backtracked or cut during the evaluation. As the evaluation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' + j = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds.
• If $s_2 \in \mathit{CutAll}(G)$ and $s_3 = \mathit{Succ}_1(s_2)$, i.e., we traverse a CutAll node, we must have $d' = 0$ since otherwise the evaluation would have backtracked or cut at least one state element of the state extension of $S_3\gamma\rho$. As S_3 contains only one state element, this is in contradiction to the condition $S_k \neq \varepsilon$. So we have $d = 0$ and the evaluation does not backtrack or cut any state element before the first state

element of the state extension of $S_3\gamma\rho$. Since this first state element of the state extension of $S_3\gamma\rho$ corresponds to the first state element of the state extension of $S_2\gamma\rho$ the evaluation does not backtrack or cut any state element before the first state element of the state extension of $S_2\gamma\rho$ and the lemma holds.

• If $s_2 \in \mathit{BacktrackSecond}(G)\backslash\{\textsc{Parallel}\}$ and $s_3 = \mathit{Succ}_2(s_2)$, we have $d = d'+1$. Furthermore, the evaluation backtracks exactly one state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the evaluation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' + 1 = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds.

• If $s_2 \in \mathit{Backtracking}(G)$ and $s_3 = \mathit{Succ}_1(s_2)$, we have $d = d'+1$. Furthermore, the evaluation backtracks exactly one state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the evaluation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' + 1 = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds.

• If $s_2 \in \mathit{VarCase}(G)$ and $s_3 = \mathit{Succ}(j, s_2)$ where s_2 has j children, then we have $d = d' + 1$. Furthermore, the evaluation backtracks exactly one state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the evaluation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' + 1 = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds.

• If $s_2 \in \mathit{Call}(G) \cup \mathit{Disjunction}(G) \cup \mathit{IfThen}(G) \cup \mathit{Repeat}(G)$ and $s_3 = \mathit{Succ}_1(s_2)$, i.e., we traverse a Call, Disjunction, IfThen, or Repeat node, then we have $d = \max(0, d'-1)$. There are two cases depending on whether $d' > 1$. If $d' > 1$, then we have $d = d' - 1$. Furthermore, the evaluation introduces exactly one additional state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$ which belongs to the first d' state elements of the state extension of $S_3\gamma\rho$. As the evaluation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' - 1 = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds. If otherwise $d' \leq 1$, then we have $d = 0$. Furthermore, the evaluation introduces exactly one additional state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the evaluation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$ and, thus, backtracks or cuts at most as many state elements as introduced from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$, it backtracks or cuts no state elements before the first state element of the state extension of $S_2\gamma\rho$ and the lemma holds.

• If $s_2 \in \mathit{IfThenElse}(G) \cup \mathit{Not}(G)$ and $s_3 = \mathit{Succ}_1(s_2)$, i.e., we traverse an IfThenElse or Not node, we have $d = \max(0, d' - 2)$. There are two cases depending on whether $d' > 2$. If $d' > 2$, then we have $d = d' - 2$. Furthermore, the evaluation introduces exactly two additional state elements from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$ which belong to the first d' state elements of the state extension of $S_3\gamma\rho$. As the evaluation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' - 2 = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds. If otherwise $d' \leq 2$, then we have $d = 0$. Furthermore, the evaluation introduces exactly two

additional state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the evaluation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$ and, thus, backtracks or cuts at most as many state elements as introduced from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$, it backtracks or cuts no state elements before the first state element of the state extension of $S_2\gamma\rho$ and the lemma holds.

• If $s_2 \in \mathit{Case}(G)$ and $s_3 = \mathit{Succ}_1(s_2)$, i.e., we traverse a CASE node, we have $d = \max(0, d' - j)$ where $S_2 = t, Q \mid S_r$ and $|\mathit{Slice}_{\mathcal{P}}(t)| = j$. There are two cases depending on whether $d' > j$. If $d' > j$, then we have $d = d' - j$. Furthermore, the evaluation introduces exactly j additional state elements from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$ which belong to the first d' state elements of the state extension of $S_3\gamma\rho$. As the evaluation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' - j = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds. If otherwise $d' \leq j$, then we have $d = 0$. Furthermore, the evaluation introduces exactly j additional state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the evaluation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$ and, thus, backtracks or cuts at most as many state elements as introduced from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$, it backtracks or cuts no state elements before the first state element of the state extension of $S_2\gamma\rho$ and the lemma holds.

• For all other cases we have $d = d'$ and the evaluation neither backtracks or cuts nor introduces state elements from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. Thus, the lemma holds.

□

Now we can show that the substitutions we use for a path (and, thus, for the rules of the TRS) correspond to the answer substitutions of the evaluations along the respective path.

Lemma C 7 (*Answer Substitutions are Instances of Path Substitutions*)
Given a path $\pi = s_1 \dots s_k$ with $s_j \notin \mathit{Inst}(G)$ for all $j \in \{1, \dots, k-1\}$ and an evaluation such that there is a variable renaming ρ and a concretization γ w.r.t. $KB_1, \dots, KB_k$ where $s_i = S_i; KB_i$ for all $i \in \{1, \dots, k\}$ and the evaluation 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 $\sigma_{\pi,0}\gamma\rho = \gamma\rho\delta$ 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 $s_1 = s_k$ and the empty answer substitution $\delta = id = \sigma_{s_1,0}$. 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 s_1 and s_2.

• If $s_1 \in \mathit{Split}(G)$ and $s_2 = \mathit{Succ}_2(s_1)$, i.e., we traverse the right child of a SPLIT node, we have $s_1 = t, Q; KB$ and $s_2 = Q\delta'; KB'$ as defined in the SPLIT rule.

By the induction hypothesis, we obtain $\sigma_{s_2 \dots s_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the evaluation from a state extension of $Q\delta'\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. For any answer substitution δ''' of the evaluation from a state extension of $(t, Q)\gamma\rho$ to a state extension of $Q\delta'\gamma\rho$ we know by Lemma A 10 that $\gamma\rho\delta''' = \delta'\gamma\rho$. Therefore, we have $\gamma\rho\delta = \gamma\rho\delta'''\delta'' = \delta'\gamma\rho\delta'' = \delta'\sigma_{s_2 \dots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Furthermore, we know that δ' is idempotent as all variables in the range of δ' are fresh. As we applied δ' to S_2 already and we know by inspection of the abstract inference rules other than Inst that only fresh variables are introduced along π, we obtain $S_k\delta' = S_k$. Hence, we have $S_k\gamma\rho\delta = S_k\gamma\rho\delta'''\delta'' = S_k\delta'\gamma\rho\delta'' = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

• If $s_1 \in \mathit{Eval}(G)$ and $s_2 = \mathit{Succ}_1(s_1)$, i.e., we traverse the left child of an Eval node, then we have $s_1 = (t, Q)^c_m \mid S; KB$ and $s_2 = B'_c\sigma', Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the Eval rule. By the induction hypothesis, we obtain $\sigma_{s_2 \dots s_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the evaluation 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 evaluation 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 the soundness proof of Eval 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 $\sigma_{\pi,0}$ starts with σ' or $\sigma|_{\mathcal{G}}$. In the first case we know by definition of $\sigma_{\pi,0}$ and Lemma C 6 that the evaluation did not backtrack the substitution σ''. Hence, we obtain $\gamma\rho\delta = \gamma\rho\sigma''\delta'' = \sigma'\gamma\rho\delta'' = \sigma'\sigma_{s_2 \dots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Additionally, we already applied σ' to S_2. As we know by inspection of all abstract inference rules other than Inst 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 $\sigma_{\pi,0}$ and Lemma C 6 that the evaluation did backtrack the substitution σ'' and we have the same answer substitution δ'' for the complete evaluation. This amounts to $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{s_2 \dots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

• If $s_1 \in \mathit{OnlyEval}(G)$ and $s_2 = \mathit{Succ}_1(s_1)$, i.e., we traverse an OnlyEval node, we have $s_1 = (t, Q)^c_m \mid S; KB$ and $s_2 = B'_c\sigma', Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the OnlyEval rule. By the induction hypothesis we obtain $\sigma_{s_2 \dots s_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the evaluation 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 evaluation 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 the soundness proof of OnlyEval 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 $\sigma_{\pi,0}$ starts with σ' or $\sigma|_{\mathcal{G}}$. In the first case we know by definition of $\sigma_{\pi,0}$ and Lemma C 6 that the evaluation did not backtrack the substitution σ''. Hence, we obtain $\gamma\rho\delta = \gamma\rho\sigma''\delta'' = \sigma'\gamma\rho\delta'' = \sigma'\sigma_{s_2 \dots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Additionally, we already applied σ' to S_2. As we know by inspection of all abstract inference rules other than Inst 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 $\sigma_{\pi,0}$ and Lemma C 6 that the evaluation did backtrack the substitution σ'' and we have the same answer substitution δ'' for the complete evaluation. This amounts to $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{s_2\ldots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

• If $s_1 \in \mathit{UnifyCase}(G)$ and $s_2 = \mathit{Succ}_1(s_1)$, i.e., we traverse the left child of a UnifyCase node, we have $s_1 = =(t_1, t_2), Q \mid S; KB$ and $s_2 = Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the UnifyCase rule. By the induction hypothesis we obtain $\sigma_{s_2\ldots s_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the evaluation from a state extension of $Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ 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 evaluation from a state extension of $(=(t_1, t_2), Q)\gamma\rho \mid S\gamma\rho$ to a state extension of $Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ we know by the soundness proof of UnifyCase 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 $\sigma_{\pi,0}$ starts with σ' or $\sigma|_{\mathcal{G}}$. In the first case we know by definition of $\sigma_{\pi,0}$ and Lemma C 6 that the evaluation did not backtrack the substitution σ''. Hence, we obtain $\gamma\rho\delta = \gamma\rho\sigma''\delta'' = \sigma'\gamma\rho\delta'' = \sigma'\sigma_{s_2\ldots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Additionally, we already applied σ' to S_2. As we know by inspection of all abstract inference rules other than Inst 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 $\sigma_{\pi,0}$ and Lemma C 6 that the evaluation did backtrack the substitution σ'' and we have the same answer substitution δ'' for the complete evaluation. This amounts to $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{s_2\ldots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

• If $s_1 \in \mathit{UnifySuccess}(G)$ and $s_2 = \mathit{Succ}_1(s_1)$, i.e., we traverse a UnifySuccess node, we have $s_1 = =(t_1, t_2), Q \mid S; KB$ and $s_2 = Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the UnifySuccess rule. By the induction hypothesis we obtain $\sigma_{s_2\ldots s_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the evaluation from a state extension of $Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ 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 evaluation from a state extension of $(=(t_1, t_2), Q)\gamma\rho \mid S\gamma\rho$ to a state extension of $Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ we know by the soundness proof of UnifySuccess 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 $\sigma_{\pi,0}$ starts with σ' or $\sigma|_{\mathcal{G}}$. In the first case we know by definition of $\sigma_{\pi,0}$ and Lemma C 6 that the evaluation did not backtrack the substitution σ''. Hence, we obtain $\gamma\rho\delta = \gamma\rho\sigma''\delta'' = \sigma'\gamma\rho\delta'' = \sigma'\sigma_{s_2\ldots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Additionally, we already applied σ' to S_2. As we know by inspection of all abstract inference rules other than Inst 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 $\sigma_{\pi,0}$ and Lemma C 6 that the evaluation did backtrack the substitution σ'' and we have the same answer substitution δ'' for the complete evaluation. This amounts to $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{s_2\ldots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

• If $s_1 \in \mathit{NoUnifyCase}(G)$ and $s_2 = \mathit{Succ}_2(s_1)$, i.e., we traverse the right child of a NoUnifyCase node, we have $s_1 = \backslash\!=\!(t_1, t_2), Q \mid S; KB$ and $s_2 = S\sigma|_{\mathcal{G}}; KB'$ as defined in the NoUnifyCase rule. By the induction hypothesis we obtain $\sigma_{s_2 \ldots s_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the evaluation from a state extension of $S\sigma|_{\mathcal{G}}\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. We know by the soundness proof of NoUnifyCase that $\gamma\rho = \sigma|_{\mathcal{G}}\gamma\rho$. As the answer substitution of the evaluation from a state extension of $(\backslash\!=\!(t_1, t_2), Q)\gamma\rho \mid S\gamma\rho$ to a state extension of $S\sigma|_{\mathcal{G}}\gamma\rho$ is empty, we know that $\delta = \delta''$ and, hence, $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{s_2 \ldots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.
• If $s_1 \in \mathit{NoUnifyFail}(G)$ and $s_2 = \mathit{Succ}_1(s_1)$, i.e., we traverse a NoUnifyFail node, we have $s_1 = \backslash\!=\!(t_1, t_2), Q \mid S; KB$ and $s_2 = S\sigma|_{\mathcal{G}}; KB'$ as defined in the NoUnifyFail rule. By the induction hypothesis we obtain $\sigma_{s_2 \ldots s_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the evaluation from a state extension of $S\sigma|_{\mathcal{G}}\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. We know by the soundness proof of NoUnifyFail that $\gamma\rho = \sigma|_{\mathcal{G}}\gamma\rho$. As the answer substitution of the evaluation from a state extension of $(\backslash\!=\!(t_1, t_2), Q)\gamma\rho \mid S\gamma\rho$ to a state extension of $S\sigma|_{\mathcal{G}}\gamma\rho$ is empty, we know that $\delta = \delta''$ and, hence, $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{s_2 \ldots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.
• If $s_1 \in \mathit{EqualsCase}(G)$ and $s_2 = \mathit{Succ}_1(s_1)$, i.e., we traverse the left child of an EqualsCase node, we have $s_1 = ==\!(t_1, t_2), Q \mid S; KB$ and $s_2 = Q\sigma \mid S\sigma; KB'$ as defined in the EqualsCase rule. By the induction hypothesis we obtain $\sigma_{s_2 \ldots s_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the evaluation from a state extension of $Q\sigma\gamma\rho \mid S\sigma\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. We know by the soundness proof of EqualsCase that $\gamma\rho = \sigma\gamma\rho$. As the answer substitution of the evaluation from a state extension of $(==\!(t_1, t_2), Q)\gamma\rho \mid S\gamma\rho$ to a state extension of $Q\sigma\gamma\rho \mid S\sigma\gamma\rho$ is empty, we know that $\delta = \delta''$ and, hence, $\gamma\rho\delta = \gamma\rho\delta'' = \sigma\gamma\rho\delta'' = \sigma\sigma_{s_2 \ldots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.
• If $s_1 \in \mathit{UnequalsCase}(G)$ and $s_2 = \mathit{Succ}_2(s_1)$, i.e., we traverse the right child of an UnequalsCase node, we have $s_1 = \backslash\!==\!(t_1, t_2), Q \mid S; KB$ and $s_2 = S\sigma; KB'$ as defined in the UnequalsCase rule. By the induction hypothesis we obtain $\sigma_{s_2 \ldots s_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the evaluation from a state extension of $S\sigma\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. We know by the soundness proof of UnequalsCase that $\gamma\rho = \sigma\gamma\rho$. As the answer substitution of the evaluation from a state extension of $(\backslash\!==\!(t_1, t_2), Q)\gamma\rho \mid S\gamma\rho$ to a state extension of $S\sigma\gamma\rho$ is empty, we know that $\delta = \delta''$ and, hence, $\gamma\rho\delta = \gamma\rho\delta'' = \sigma\gamma\rho\delta'' = \sigma\sigma_{s_2 \ldots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. 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 evaluation 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 $\sigma_{s_2 \ldots s_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the evaluation 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 $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{s_2 \ldots s_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$ and $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

□

Now, using the preceding results, we can simulate the traversal of a tree path by the rewrite rules of the corresponding TRS. To this end, we first need the notions of a relative derivation length (in order to show that the function value of $irc_{\mathcal{R}(G_s),\mathcal{R}(G)}$ for a node s in a derivation graph G is at least as high as the length of a certain derivation) and a full subgraph.

Definition C 8 (*Relative Derivation Length*)
Let $\mathcal{R}$ be a TRS, $\mathcal{R}' \subseteq \mathcal{R}$, and $d = (t \to_{\ell_1 \to r_1} t_1 \to_{\ell_2 \to r_2} \dots \to_{\ell_k \to r_k} t_k)$ be a derivation w.r.t. $\mathcal{R}$. Then the *relative length* $RelLength_{\mathcal{R}'}(d)$ is $|\{i \mid \ell_i \to r_i \in \mathcal{R}'\}|$.

Moreover, for a composable derivation graph G which is decomposed into the subgraphs $G_1, \dots, G_k$ according to Def. 10, a node s in G which is the root of G, a multiplicative Split node in G or a successor of a multiplicative Split node in G, and derivations $d_1, \dots, d_k$ such that there is a function f mapping each successor of a multiplicative Split node and the root to an index of one subgraph, we define

$$cplx_s(d_1, \dots, d_k) =$$

$$\begin{cases} cplx_{Succ_1(s)}(d_1, \dots, d_k) \cdot cplx_{Succ_2(s)}(d_1, \dots, d_k), & \text{if } s \in mults(G) \\ RelLength_{\mathcal{R}(G_s)}(d_{f(s)}) + \Sigma_{s' \in mults(G) \cap G_s}\ cplx_{s'}(d_1, \dots, d_k), & \text{otherwise.} \end{cases}$$

Definition C 9 (*Full Subgraph*)
Let G be a derivation graph with nodes V and edges E (i.e., $G = (V, E)$) and let $s \in V$. Then we define the full subgraph of G at node s *as the minimal graph $G_s = (V_s, E_s)$ where $s \in V_s$ and whenever $s_1 \in V_s$ and $(s_1, s_2) \in E$, then $s_2 \in V_s$ and $(s_1, s_2) \in E_s$.*

We first show that subgraphs without Inst edges (i.e., acyclic subgraphs) can only represent evaluations whose length is bounded by a constant.

Lemma C 10 (*Constant (Sub-)Graphs*)
Let G be a derivation graph with k nodes, $|Split(G)| = j \leq k$, and $Inst(G) = \varnothing$. Then for all $Q \in \mathcal{CON}(root(G))$, the evaluation of Q takes at most k^{2^j} steps. Moreover, any tree path π in G has $size(\pi) \leq k^{2^j}$.

Proof
Let $Q \in \mathcal{CON}(root(G))$. We perform the proof of the lemma by induction over the number k of nodes in G. Note that $size(\pi)$ is always at least as big as $numOfSuccesses(\pi)$.

If $k = 1$, then we must have $root(G) = \varepsilon; KB$ and $Q = \varepsilon$ whose evaluation takes $0 < 1 = 1^{2^0}$ steps. Moreover, the only tree path in G is $root(G)$ which has $size(root(G)) = 1 = 1^{2^0}$.

If $k > 1$, then we assume that the evaluation of Q takes more than 0 steps and π consists of at least two nodes, since otherwise, there is nothing to show. We perform a case analysis over $s = root(G)$.

• If $root(G) \in Parallel(G)$, then we have $root(G) = S_1 \mid S_2; KB$ and $Q = Q_1 \mid Q_2$ with $Q_1 \neq \varepsilon$ and $Q_2 \neq \varepsilon$ such that $Q_1 \in \mathcal{CON}(Succ_1(root(G)))$ and $Q_2 \in$

$\mathcal{CON}(Succ_2(root(G)))$. Therefore, we have $k > 4$. As $root(G)$ is not reachable from its successors and no successor of $root(G)$ can reach the other successor, we can consider the respective subgraphs G_1 and G_2 at the successors of $root(G)$. Let G_1 contain k' nodes and $|Split(G_1)| = j'$. Then we have $0 < k' < k-1$, G_2 has $k-k'-1$ nodes, and $|Split(G_2)| = j - j' \geq 0$. Thus, we can use the induction hypothesis to obtain that the evaluation of Q_1 takes at most $(k')^{2^{j'}}$ steps, any tree path π_1 in G_1 has $size(\pi_1) \leq (k')^{2^{j'}}$, the evaluation of Q_2 takes at most $(k-k'-1)^{2^{j-j'}}$ steps, and any tree path π_2 in G_2 has $size(\pi_2) \leq (k-k'-1)^{2^{j-j'}}$. Furthermore, we show two little helping propositions. First $(*)$, for all $a, b > 1$, we have $a + b \leq a \cdot b$. This is shown by induction over a. If $a = 2$, then we have $2 + b \overset{2 \leq b}{\leq} b + b = 2 \cdot b$. If $a > 2$, then we have

$$\begin{array}{rcl} a+b & = & 1+(a-1+b) \\ & \overset{\text{induction hypothesis}}{\leq} & 1+(a-1)\cdot b \\ & = & a \cdot b - b + 1 \\ & \overset{b>1}{<} & a \cdot b. \end{array}$$

Second $(**)$, for all a, b, c with $0 \leq b \leq a$ and $c > 0$, we have $b + (a-b)^c \leq a^c$. We show this proposition by induction over c. If $c = 1$, then we have $b + (a-b) = a$. If $c > 1$, we first consider the case $a = b = 0$. Then we obtain $0 + (0-0)^c = 0 = 0^c$. Otherwise, we have $a > 0$ and obtain

$$\begin{array}{rcl} b+(a-b)^c & = & b+((b+(a-b)^{c-1})-b)\cdot(a-b) \\ & \overset{\text{induction hypothesis}}{\leq} & b+(a^{c-1}-b)\cdot(a-b) \\ & = & b+a^{c-1}\cdot(a-b)-b\cdot(a-b) \\ & = & b+a^{c-1}\cdot a-a^{c-1}\cdot b-b\cdot a+b^2 \\ & = & a^c+b-b\cdot a^{c-1}-b\cdot a+b^2 \\ & = & a^c+b\cdot(1-a^{c-1}-a+b) \\ & \overset{c>1,a>0}{\leq} & a^c+b\cdot(1-1-a+b) \\ & = & a^c+b\cdot(b-a) \\ & \overset{b\leq a}{\leq} & a^c+b\cdot 0 \\ & = & a^c. \end{array}$$

Now we continue with the proof. If $j = j' = 0$, then we obtain that the evaluation of Q cannot take more than $k' + (k-k'-1) = k-1 < k$ steps. Moreover, we have $size(\pi) \leq k' + (k-k'-1) + 1 = k$. Now consider $j > 0$. If $j' = 0$, then we know that the evaluation of Q has at most length

$$\begin{array}{rcl} k'+(k-k'-1)^{2^j} & \overset{0<k'<k-1,(**)}{\leq} & (k-1)^{2^j} \\ & < & k^{2^j}. \end{array}$$

Moreover, we have

$$\begin{array}{lll} size(\pi) & \leq & k' + (k - k' - 1)^{2^j} + 1 \\ & \overset{0<k'<k-1,(**)}{\leq} & (k-1)^{2^j} + 1 \\ & \leq & k^{2^j}. \end{array}$$

If $j' = j$, then we obtain a length of at most

$$\begin{array}{lll} (k')^{2^j} + (k - k' - 1) & \overset{0<k'<k-1,(**)}{\leq} & (k' + k - k' - 1)^{2^j} \\ & = & (k-1)^{2^j} \\ & < & k^{2^j} \end{array}$$

and

$$\begin{array}{lll} size(\pi) & \leq & (k')^{2^j} + (k - k' - 1) + 1 \\ & \overset{0<k'<k-1,(**)}{\leq} & (k' + k - k' - 1)^{2^j} + 1 \\ & = & (k-1)^{2^j} + 1 \\ & \leq & k^{2^j}. \end{array}$$

Finally, we have $0 < j' < j$ and know that the evaluation of Q cannot take more than

$$\begin{array}{lll} (k')^{2^{j'}} + (k - k' - 1)^{2^{j-j'}} & \overset{0<k'<k-1}{\leq} & (k-2)^{2^{j'}} + (k-2)^{2^{j-j'}} \\ & \overset{k>4,(*)}{\leq} & (k-2)^{2^{j'}} \cdot (k-2)^{2^{j-j'}} \\ & = & (k-2)^{2^{j'} + 2^{j-j'}} \\ & \overset{0<j'<j,(*)}{\leq} & (k-2)^{2^{j'} \cdot 2^{j-j'}} \\ & = & (k-2)^{2^{j'+j-j'}} \\ & = & (k-2)^{2^j} \\ & < & k^{2^j} \end{array}$$

steps. Furthermore, we have

$$\begin{array}{lll} size(\pi) & \leq & (k')^{2^{j'}} + (k - k' - 1)^{2^{j-j'}} + 1 \\ & \overset{0<k'<k-1}{\leq} & (k-2)^{2^{j'}} + (k-2)^{2^{j-j'}} + 1 \\ & \overset{k>4,(*)}{\leq} & (k-2)^{2^{j'}} \cdot (k-2)^{2^{j-j'}} + 1 \\ & = & (k-2)^{2^{j'} + 2^{j-j'}} + 1 \\ & \overset{0<j'<j,(*)}{\leq} & (k-2)^{2^{j'} \cdot 2^{j-j'}} + 1 \\ & = & (k-2)^{2^{j'+j-j'}} + 1 \\ & = & (k-2)^{2^j} + 1 \\ & \leq & k^{2^j}. \end{array}$$

• If $root(G) \in Split(G)$, then we have $root(G) = t', Q'; KB$ and $Q = t, Q''$ with $t \neq \varepsilon$ and $Q'' \neq \varepsilon$ such that $t \in \mathcal{CON}(Succ_1(root(G)))$ and $Q'' \in \mathcal{CON}(Succ_2(root(G)))$. Therefore, we have $k > 4$. As $root(G)$ is not reachable from its successors and no successor of $root(G)$ can reach the other successor, we can consider the respective subgraphs G_1 and G_2 at the successors of $root(G)$. Let G_1 contain k' nodes and $|Split(G_1)| = j'$. Then we have $0 < k' < k - 1$, $j' < j$, G_2 has $k - k' - 1$ nodes, and $|Split(G_2)| = j - j' - 1$. Thus, we can use the induction hypothesis to obtain that the evaluation of t takes at most $(k')^{2^{j'}}$ steps, any tree path π_1 in G_1 has $size(\pi_1) \leq (k')^{2^{j'}}$, the evaluation of Q'' takes at most $(k - k' - 1)^{2^{j-j'-1}}$ steps, and any tree path π_2 in G_2 has $size(\pi_2) \leq (k - k' - 1)^{2^{j-j'-1}}$. If $j' = 0$, then we obtain that the evaluation of Q cannot take more than

$$\begin{array}{lcl} k' \cdot (k - k' - 1)^{2^{j-1}} & \overset{0<k'<k-1}{<} & k \cdot k^{2^{j-1}} \\ & = & k^{2^{j-1}+1} \\ & \overset{j>0}{\leq} & k^{2^j} \end{array}$$

steps. Moreover, we have

$$\begin{array}{lcl} size(\pi) & \leq & k' \cdot (k - k' - 1)^{2^{j-1}} + 1 \\ & \overset{0<k'<k-1}{\leq} & k \cdot k^{2^{j-1}} \\ & = & k^{2^{j-1}+1} \\ & \overset{j>0}{\leq} & k^{2^j}. \end{array}$$

If $j' = j - 1$, then we obtain a length of

$$\begin{array}{lcl} (k')^{2^{j-1}} \cdot (k - k' - 1) & \overset{0<k'<k-1}{<} & k^{2^{j-1}} \cdot k \\ & = & k^{2^{j-1}+1} \\ & \overset{j>0}{\leq} & k^{2^j} \end{array}$$

and

$$\begin{array}{lcl} size(\pi) & \leq & (k')^{2^{j-1}} \cdot (k - k' - 1) + 1 \\ & \overset{0<k'<k-1}{\leq} & k^{2^{j-1}} \cdot k \\ & = & k^{2^{j-1}+1} \\ & \overset{j>0}{\leq} & k^{2^j}. \end{array}$$

Finally, if $0 < j' < j-1$, then we know that the evaluation of Q takes at most

$$\begin{array}{lll}
(k')^{2^{j'}} \cdot (k-k'-1)^{2^{j-j'-1}} & \overset{0<k'\leq k-1}{\leq} & (k-2)^{2^{j'}} \cdot (k-2)^{2^{j-j'-1}} \\
& = & (k-2)^{2^{j'}+2^{j-j'-1}} \\
& \overset{0<j'<j-1,(*)}{\leq} & (k-2)^{2^{j'}\cdot 2^{j-j'-1}} \\
& = & (k-2)^{2^{j'+j-j'-1}} \\
& < & (k-2)^{2^j} \\
& < & k^{2^j}
\end{array}$$

steps and we have

$$\begin{array}{lll}
size(\pi) & \leq & (k')^{2^{j'}} \cdot (k-k'-1)^{2^{j-j'-1}} + 1 \\
& \overset{0<k'\leq k-1}{\leq} & (k-2)^{2^{j'}} \cdot (k-2)^{2^{j-j'-1}} + 1 \\
& = & (k-2)^{2^{j'}+2^{j-j'-1}} + 1 \\
& \overset{0<j'<j-1,(*)}{\leq} & (k-2)^{2^{j'}\cdot 2^{j-j'-1}} + 1 \\
& = & (k-2)^{2^{j'+j-j'-1}} + 1 \\
& \leq & (k-2)^{2^j} \\
& < & k^{2^j}.
\end{array}$$

• Otherwise, the second state in the evaluation of Q is represented by one of the successors of $root(G)$. As $root(G)$ is not reachable from its successors, we can consider the subgraph at this successor which has at most $k-1$ nodes. We use the induction hypothesis to obtain that the remaining evaluation takes at most $(k-1)^{2^j}$ steps. Together with the first step, we obtain that the evaluation takes at most $(k-1)^{2^j} + 1 \overset{k>1}{\leq} k^{2^j}$ steps. For π we also know that its root can only have one successor as branching is only possible in a tree path at PARALLEL or SPLIT nodes. Thus we have $size(\pi) \leq (k-1)^{2^j} + 1 \overset{k>1}{\leq} k^{2^j}$.

□

We need to be able to simulate arbitrary answer substitutions. This is shown in the following lemma.

Lemma C 11 (*Simulating Answers*)
Let G be a derivation graph and $Q = S\gamma \in \mathcal{CON}(root(G))$ where Q does not contain the goal □. Then we have $ren^{in}(root(G))\gamma \to^*_{\mathcal{R}(G)} ren^{out}(root(G), i)\gamma\delta$ if the i-th goal in Q produces the answer substitution δ in finitely many steps.

Proof
We consider a finite prefix with length ℓ of the evaluation of Q. We perform the proof by induction over the lexicographic combination of first ℓ and second the edge relation of G restricted to INST, PARALLEL, and SPLIT edges. Note that this induction relation is indeed well founded as the sum of the number of terms and

goals in a state is strictly decreased by applying the rules PARALLEL or SPLIT while it stays equal by applying the INST rule. So every infinite sequence of rule applications with only these rules must eventually only use the INST rule. This is in contradiction to the requirement that derivation graphs do not contain cycles consisting of INST edges only.

If $\ell = 0$, then the prefix of the evaluation does not produce any answer substitution and the lemma trivially holds.

If $\ell > 0$, then we perform a case analysis over $s = root(G)$.

- If $s \in Inst(G)$, then we consider $Succ_1(s)$ and $\gamma' = \mu\gamma$ instead of s and γ where $Q = Succ_1(s)\gamma'$ (the latter follows from the soundness proof of INST). By the induction hypothesis, we obtain that $ren^{in}(Succ_1(s))\gamma' \to^*_{\mathcal{R}(G)} ren^{out}(Succ_1(s), i)\gamma'\delta$. By $\gamma' = \mu\gamma$, $ren^{in}(s) = ren^{in}(Succ_1(s))\mu$, and $ren^{out}(s, i) = ren^{out}(Succ_1(s), i)\mu$, we conclude $ren^{in}(s)\gamma \to^*_{\mathcal{R}(G)} ren^{out}(s, i)\gamma\delta$.
- If $s \in Parallel(G)$, then we have the rules

$$\begin{gathered} ren^{in}(s) \\ \to \\ \mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s)), \mathcal{V}(ren^{in}(s))), \end{gathered}$$

$$\begin{gathered} ren^{in}(s) \\ \to \\ \mathsf{u}_{s,Succ_2(s)}(ren^{in}(Succ_2(s)), \mathcal{V}(ren^{in}(s))) \end{gathered}$$

in $\mathcal{R}(G)$. Moreover, we have for all $j \in \{1, \ldots, k_1\}$ where $Succ_1(s)$ has k_1 goals and the j-th goal is no scope marker the rule

$$\begin{gathered} \mathsf{u}_{s,Succ_1(s)}(ren^{out}(Succ_1(s), j), \mathcal{V}(ren^{in}(s))) \\ \to \\ ren^{out}(s, j) \end{gathered}$$

in $\mathcal{R}(G)$ and for all $j \in \{1, \ldots, k_2\}$ where $Succ_2(s)$ has k_2 goals and the j-th goal is no scope marker the rule

$$\begin{gathered} \mathsf{u}_{s,Succ_2(s)}(ren^{out}(Succ_2(s), j), \mathcal{V}(ren^{in}(s))) \\ \to \\ ren^{out}(s, j + k_1) \end{gathered}$$

in $\mathcal{R}(G)$. There are two cases.
If the i-th goal is contained in $Succ_1(s)$, then we rewrite $ren^{in}(s)\gamma$ to $\mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s)), \mathcal{V}(ren^{in}(s)))\gamma$. By the induction hypothesis, we obtain $ren^{in}(Succ_1(s))\gamma \to^*_{\mathcal{R}(G)} ren^{out}(Succ_1(s), i)\gamma\delta$. Thus, we have

$$\begin{gathered} \mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s)), \mathcal{V}(ren^{in}(s)))\gamma \\ \to^*_{\mathcal{R}(G)} \\ \mathsf{u}_{s,Succ_1(s)}(ren^{out}(Succ_1(s), i), \mathcal{V}(ren^{in}(s)))\gamma\delta \\ \to_{\mathcal{R}(G)} \\ ren^{out}(s, i)\gamma\delta. \end{gathered}$$

If the i-th goal of s is contained in $Succ_2(s)$, let i' be the index of this goal in $Succ_2(s)$, i.e., $i = k_1 + i'$. Then we rewrite $ren^{in}(s)\gamma$ to $\mathsf{u}_{s,Succ_2(s)}(ren^{in}(Succ_2(s)), \mathcal{V}(ren^{in}(s)))\gamma$. By the induction hypothesis, we obtain $ren^{in}(Succ_2(s))\gamma \to^*_{\mathcal{R}(G)} ren^{out}(Succ_2(s), i')\gamma\delta$. Thus, we have

$$\begin{gathered}\mathsf{u}_{s,Succ_2(s)}(ren^{in}(Succ_2(s)), \mathcal{V}(ren^{in}(s)))\gamma \\ \to^*_{\mathcal{R}(G)} \\ \mathsf{u}_{s,Succ_2(s)}(ren^{out}(Succ_2(s), i'), \mathcal{V}(ren^{in}(s)))\gamma\delta \\ \to_{\mathcal{R}(G)} \\ ren^{out}(s, i' + k_1)\gamma\delta = ren^{out}(s, i)\gamma\delta.\end{gathered}$$

- If $s \in Split(G)$, we have $S = t, Q'$ and the rules

$$\begin{gathered}ren^{in}(s)\delta' \\ \to \\ \mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s)), \mathcal{V}(ren^{in}(s)))\delta',\end{gathered}$$

$$\begin{gathered}\mathsf{u}_{s,Succ_1(s)}(ren^{out}(Succ_1(s), 1), \mathcal{V}(ren^{in}(s)))\delta' \\ \to \\ \mathsf{u}_{Succ_1(s),Succ_2(s)}(ren^{in}(Succ_2(s)), (\mathcal{V}(ren^{in}(s)) \cup \mathcal{V}(ren^{out}(Succ_1(s), 1)))\delta'), \text{ and}\end{gathered}$$

$$\begin{gathered}\mathsf{u}_{Succ_1(s),Succ_2(s)}(ren^{out}(Succ_2(s), 1), (\mathcal{V}(ren^{in}(s)) \cup \mathcal{V}(ren^{out}(Succ_1(s), 1)))\delta') \\ \to \\ ren^{out}(s, 1)\delta'\end{gathered}$$

where δ' is the substitution associated to s in $\mathcal{R}(G)$. Since δ is an answer substitution for the whole goal, there must be a corresponding answer substitution δ'' for $t\gamma$. By Lemma A 10 we know that there is a γ' with $\gamma\delta'' = \delta'\gamma'$. So we can rewrite $ren^{in}(s)\gamma$ to $\mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s)), \mathcal{V}(ren^{in}(s)))\delta'\gamma'$. By the induction hypothesis and Lemma A 10, we obtain $ren^{in}(Succ_1(s))\delta'\gamma' \to^*_{\mathcal{R}(G)} ren^{out}(Succ_1(s), 1)\delta'\gamma'$. Thus, we have

$$\begin{gathered}\mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s)), \mathcal{V}(ren^{in}(s)))\delta'\gamma' \\ \to^*_{\mathcal{R}(G)} \\ \mathsf{u}_{s,Succ_1(s)}(ren^{out}(Succ_1(s), 1), \mathcal{V}(ren^{in}(s)))\delta'\gamma' \\ \to_{\mathcal{R}(G)} \\ \mathsf{u}_{Succ_1(s),Succ_2(s)}(ren^{in}(Succ_2(s)), (\mathcal{V}(ren^{in}(s)) \cup \mathcal{V}(ren^{out}(Succ_1(s), 1)))\delta')\gamma'.\end{gathered}$$

Again, by the induction hypothesis we obtain $ren^{in}(Succ_2(s))\gamma' \to^*_{\mathcal{R}(G)} ren^{out}(Succ_2(s), 1)\delta'''$ where $\delta = \delta''\delta'''$. Together, we obtain

$$\begin{gathered}\mathsf{u}_{Succ_1(s),Succ_2(s)}(ren^{in}(Succ_2(s)), (\mathcal{V}(ren^{in}(s)) \cup \mathcal{V}(ren^{out}(Succ_1(s), 1)))\delta')\gamma' \\ \to^*_{\mathcal{R}(G)} \\ \mathsf{u}_{Succ_1(s),Succ_2(s)}(ren^{out}(Succ_2(s), 1), (\mathcal{V}(ren^{in}(s)) \cup \mathcal{V}(ren^{out}(Succ_1(s), 1)))\delta')\gamma'\delta''' \\ \to_{\mathcal{R}(G)} \\ ren^{out}(s, 1)\delta'\gamma'\delta''' = ren^{out}(s, 1)\gamma\delta''\delta''' = ren^{out}(s, 1)\gamma\delta.\end{gathered}$$

- Otherwise, we consider the nodes from s to the first node $s' \in Suc(G) \cup Inst(G) \cup Parallel(G) \cup Split(G) \cup Succ_1(Inst(G))$ reached by the evaluation of Q. If no such

node as s' exists, then Q cannot produce any answer substitutions and the lemma trivially holds. Thus, we consider the case that such a node s' exists. Then we directly have that the path from s to s' is a connection path π_c in G. Thus we have the rule $ren^{in}(s)\sigma_{\pi_c,0} \to \mathsf{u}_{s,s'}(ren^{in}(s'), \mathcal{V}(ren^{in}(s)))\sigma_{\pi_c,0})$ in $\mathcal{R}(G)$ and for all $j \in \{1,\ldots,k\}$ where s' has k goals and the j-th goal is no scope marker, we have the rule

$$\begin{gathered}\mathsf{u}_{s,s'}(ren^{out}(s',j), \mathcal{V}(ren^{in}(s)))\sigma_{\pi_c,j-1}) \\ \to \\ ren^{out}(s, skip(\pi_c, j))\sigma_{\pi_c,j-1}\end{gathered}$$

in $\mathcal{R}(G)$. By Lemma C 7, we know that $\sigma_{\pi_c,0}$ is an instance of the answer substitution along π_c. In particular, we know that $ren^{in}(s)\sigma_{\pi_c,0}$ matches t by γ and we can rewrite t to $\mathsf{u}_{s,s'}(ren^{in}(s'), \mathcal{V}(ren^{in}(s)))\sigma_{\pi_c,0})\gamma$. For the i-th goal of s producing an answer substitution, there must be a corresponding j-th goal in s' producing an answer substitution δ'. We use the induction hypothesis for s' to obtain that the term $ren^{in}(s')\gamma$ rewrites to the term $ren^{out}(s',j)\gamma\delta'$. Thus, we have

$$\begin{aligned}\mathsf{u}_{s,s'}(ren^{in}(s'), \mathcal{V}(ren^{in}(s)))\sigma_{\pi_c,0})\gamma &\to^*_{\mathcal{R}(G)} \\ \mathsf{u}_{s,s'}(ren^{out}(s',j), \mathcal{V}(ren^{in}(s)))\sigma_{\pi_c,0})\gamma\delta' &\to_{\mathcal{R}(G)} ren^{out}(s,i)\gamma\delta'\delta''\end{aligned}$$

where $\delta = \delta'\delta''$.

□

As we can simulate answer substitutions, we can also simulate tree paths and, hence, evaluations.

Lemma C 12 (*Simulating Tree Paths*)
Let G be a decomposable derivation graph which is decomposed into k subgraphs according to Def. 10, $root(G) = s = S; KB$, $\mathcal{R}(G) \neq \varnothing$, and π be a tree path in G for a prefix of the evaluation of a query $Q = S\gamma \in \mathcal{CON}(s)$ with the properties from Lemma C 4 for the tree path π_b. Then the term $t = ren^{in}(s)\gamma$ is a basic term w.r.t. $\mathcal{R}(G)$ and starts k derivations $d_1 = (t \to_{\mathcal{R}(G)} \ldots \to_{\mathcal{R}(G)} t_1), \ldots, d_k = (t \to_{\mathcal{R}(G)} \ldots \to_{\mathcal{R}(G)} t_k)$ such that $size(\pi) \in \mathcal{O}(cplx_s(d_1,\ldots,d_k))$.

Proof
We first show that t is a basic term w.r.t. $\mathcal{R}(G)$, i.e., there is a rule $\mathsf{f}(a_1,\ldots,a_j) \to r \in \mathcal{R}(G)$ with $\mathsf{f} = root(t)$ and t does not contain any defined symbols at a non-empty position. The latter is obviously true since in $\mathcal{R}(G)$, we only used fresh function symbols. So Q cannot contain them and, hence, γ cannot replace the variables in $ren^{in}(root(G))$ by terms with such symbols. For a rule as desired to be in $\mathcal{R}(G)$, we need a connection path starting in $root(G)$ or $root(G)$ being a Parallel or a Split node. Then the other condition is easily seen by inspection of *ConnectionRules*, *ParallelRules*, and *SplitRules*. So assume we have no connection path starting in $root(G)$ and $root(G)$ is neither a Parallel nor a Split node. Then there is no node in $Inst(G) \cup Suc(G) \cup Parallel(G) \cup Split(G) \cup Succ_1(Inst(G))$ which is reachable from $root(G)$. This again means that no start node for a connection

path and no PARALLEL or SPLIT node exists in G. Thus, we would have $\mathcal{R}(G) = \varnothing$ which contradicts our assumptions and, hence, proves that t is a basic term w.r.t. $\mathcal{R}(G)$. We are left to show that t starts k derivations $d_1, \ldots, d_k$ w.r.t. $\mathcal{R}(G)$ such that $size(\pi) \in \mathcal{O}(cplx_{root(G)}(d_1, \ldots, d_k))$. We perform the proof of this proposition by induction over $size(\pi)$.

For $size(\pi) = 0$, we trivially have $size(\pi) \in \mathcal{O}(1) \subseteq \mathcal{O}(1 + cplx_{root(G)}(d_1, \ldots, d_k))$ $= \mathcal{O}(cplx_{root(G)}(d_1, \ldots, d_k))$.

For $size(\pi) > 0$, we perform a case analysis on the first node $s = root(G)$ in π.

- If $s \in Inst(G)$, we know that $ren^{in}(s) = ren^{in}(Succ_1(s))\mu$ where μ is associated to s. Thus, we have $t = ren^{in}(s)\gamma = ren^{in}(Succ_1(s))\mu\gamma$. By Lemma A 11, we know that $Q \in \mathcal{CON}(Succ_1(s))$. Thus, we can consider the full subgraph G' at $Succ_1(s)$ instead of G for which the root of the only direct subtree π' of π is the root node and apply the induction hypothesis to obtain that $size(\pi') \in \mathcal{O}(cplx_{root(G')}(d_1, \ldots, d_k))$. We conclude that $size(\pi) = size(\pi') + 1 \in \mathcal{O}(cplx_{root(G')}(d_1, \ldots, d_k) + 1) = \mathcal{O}(cplx_{root(G')}(d_1, \ldots, d_k)) \subseteq \mathcal{O}(cplx_{root(G)}(d_1, \ldots, d_k))$.
- If $s \in Parallel(G)$, we have the rules

$$ren^{in}(s) \to \mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s)), \mathcal{V}(ren^{in}(s)))$$

and

$$ren^{in}(s) \to \mathsf{u}_{s,Succ_2(s)}(ren^{in}(Succ_2(s)), \mathcal{V}(ren^{in}(s)))$$

in $\mathcal{R}(G)$. Let π_1 denote the first direct subtree of π and π_2 denote the second one if it exists. There are two cases.

If π only has one direct subtree or if $size(\pi_1) \geq size(\pi_2)$, then we have $ren^{in}(s)\gamma \to_{\mathcal{R}(G)} \mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s)), \mathcal{V}(ren^{in}(s)))\gamma$. Thus, we use the induction hypothesis for $Succ_1(s)$ to obtain k derivations $d'_1, \ldots, d'_k$ with $size(\pi_1) \in \mathcal{O}(cplx_{Succ_1(s)}(d'_1, \ldots, d'_k))$. Adding the first rewrite step to all k derivations, we obtain the derivations $d_1, \ldots, d_k$ with $size(\pi) \leq 1 + 2 \cdot size(\pi_1) \in \mathcal{O}(1 + 2 \cdot cplx_{Succ_1(s)}(d'_1, \ldots, d'_k)) = \mathcal{O}(cplx_{Succ_1(s)}(d'_1, \ldots, d'_k)) \subseteq \mathcal{O}(cplx_s(d_1, \ldots, d_k))$.

If π has two direct subtrees and $size(\pi_1) < size(\pi_2)$, then we have $ren^{in}(s)\gamma \to_{\mathcal{R}(G)} \mathsf{u}_{s,Succ_2(s)}(ren^{in}(Succ_2(s)), \mathcal{V}(ren^{in}(s)))\gamma$. Thus, we use the induction hypothesis for $Succ_2(s)$ to obtain k derivations $d'_1, \ldots, d'_k$ with $size(\pi_2) \in \mathcal{O}(cplx_{Succ_2(s)}(d'_1, \ldots, d'_k))$. Adding the first rewrite step to all k derivations, we obtain the derivations $d_1, \ldots, d_k$ with $size(\pi) \leq 1 + 2 \cdot size(\pi_2) \in \mathcal{O}(1 + 2 \cdot cplx_{Succ_2(s)}(d'_1, \ldots, d'_k)) = \mathcal{O}(cplx_{Succ_2(s)}(d'_1, \ldots, d'_k)) \subseteq \mathcal{O}(cplx_s(d_1, \ldots, d_k))$.
- If $s \in Split(G)$, then we have the rules

$$ren^{in}(s)\delta \to \mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s))\delta, \mathcal{V}(ren^{in}(s))\delta),$$

$$\mathsf{u}_{s,Succ_1(s)}(ren^{out}(Succ_1(s), 1)\delta, \mathcal{V}(ren^{in}(s))\delta)$$

$$\to$$
$$\mathsf{u}_{Succ_1(s),Succ_2(s)}(ren^{in}(Succ_2(s)), (\mathcal{V}(ren^{in}(s)) \cup \mathcal{V}(ren^{out}(Succ_1(s),1)))\delta), \text{ and}$$
$$\mathsf{u}_{Succ_1(s),Succ_2(s)}(ren^{out}(Succ_2(s),1), (\mathcal{V}(ren^{in}(s)) \cup \mathcal{V}(ren^{out}(Succ_1(s),1)))\delta)$$
$$\to$$
$$ren^{out}(s,1)\delta$$

where δ is the substitution associated to s in $\mathcal{R}(G)$. Let π_1 denote the first direct subtree of π and π_2 denote the second direct subtree of π if it exists. By Lemma C 7 and Lemma A 10, we know that there is a γ' such that $ren^{in}(s)\gamma \to_{\mathcal{R}(G)} \mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s))\delta, \mathcal{V}(ren^{in}(s))\delta)\gamma'$.

If π has only one direct subtree, then it must start with $Succ_1(s)$ and we use the induction hypothesis on it and the term $ren^{in}(Succ_1(s)\delta$ to obtain k derivations $d'_1, \ldots, d'_k$ such that $size(\pi_1) \in \mathcal{O}(cplx_{Succ_1(s)}(d'_1, \ldots, d'_k))$. Adding the first rewrite step to all k derivations, we obtain the derivations $d_1, \ldots, d_k$ with $size(\pi) \leq 1 + 2 \cdot size(\pi_1) \in \mathcal{O}(1 + 2 \cdot cplx_{Succ_1(s)}(d'_1, \ldots, d'_k)) = \mathcal{O}(cplx_{Succ_1(s)}(d'_1, \ldots, d'_k)) \subseteq \mathcal{O}(cplx_s(d_1, \ldots, d_k))$.

If $size(\pi_1) \geq numOfSuccesses(\pi_1) \cdot size(\pi_2)$, the proof is analogous to the case where π has only one direct subtree.

So let $size(\pi_1) < numOfSuccesses(\pi_1) \cdot size(\pi_2)$. By Lemma C 11 we obtain a derivation $ren^{in}(Succ_1(s))\delta\gamma' \to^*_{\mathcal{R}(G)} ren^{out}(Succ_1(s),1)\delta\gamma'$. Hence, we have

$$\mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s))\delta, \mathcal{V}(ren^{in}(s))\delta)\gamma'$$
$$\to^*_{\mathcal{R}(G)}$$
$$\mathsf{u}_{s,Succ_1(s)}(ren^{out}(Succ_1(s),1)\delta, \mathcal{V}(ren^{in}(s))\delta)\gamma'$$
$$\to_{\mathcal{R}(G)}$$
$$\mathsf{u}_{Succ_1(s),Succ_2(s)}(ren^{in}(Succ_2(s)), (\mathcal{V}(ren^{in}(s)) \cup \mathcal{V}(ren^{out}(Succ_1(s),1)))\delta)\gamma'.$$

If s is not multiplicative, i.e., $numOfSuccesses(\pi_1)$ is bounded by a constant c, then we use the induction hypothesis on $Succ_2(s)$ and the term $ren^{in}(Succ_2(s))\gamma'$ to obtain k derivations $d'_1, \ldots, d'_k$ with $size(\pi_2) \in \mathcal{O}(cplx_{Succ_2(s)}(d'_1, \ldots, d'_k))$. Adding the derivation from $ren^{in}(s)\gamma$ to $\mathsf{u}_{Succ_1(s),Succ_2(s)}(ren^{in}(Succ_2(s)), (\mathcal{V}(ren^{in}(s)) \cup \mathcal{V}(ren^{out}(Succ_1(s),1)))\delta)\gamma'$ to all k derivations, we obtain the derivations $d_1, \ldots, d_k$ with $size(\pi) \leq 1 + 2 \cdot c \cdot size(\pi_2) \in \mathcal{O}(1 + 2 \cdot c \cdot cplx_{Succ_2(s)}(d'_1, \ldots, d'_k)) = \mathcal{O}(cplx_{Succ_2(s)}(d'_1, \ldots, d'_k)) \subseteq \mathcal{O}(cplx_s(d_1, \ldots, d_k))$.

If s is multiplicative, then we use the induction hypothesis on both $Succ_1(s)$ and $Succ_2(s)$ to obtain the derivations $d^a_1, \ldots, d^a_k$ for $ren^{in}(Succ_1(s))\delta\gamma'$ and $d^b_1, \ldots, d^b_k$ for $ren^{in}(Succ_2(s))\gamma'$ with $size(\pi_1) \in \mathcal{O}(cplx_{Succ_1(s)}(d^a_1, \ldots, d^a_k))$ and $size(\pi_2) \in \mathcal{O}(cplx_{Succ_2(s)}(d^b_1, \ldots, d^b_k))$. As G is decomposed into k subgraphs, only $k' < k$ subgraphs can be reached from $Succ_1(s)$. We choose those k' indices corresponding to these subgraphs to take k' derivations from $d^a_1, \ldots, d^a_k$ and the other ones from $d^b_1, \ldots, d^b_k$ after adding the derivation from

$$ren^{in}(s)\gamma$$
to
$$\mathsf{u}_{s,Succ_1(s)}(ren^{in}(Succ_1(s))\delta, \mathcal{V}(ren^{in}(s))\delta)\gamma'$$

to the k' derivations and the derivation from

$$ren^{in}(s)\gamma$$
to
$$\mathsf{u}_{Succ_1(s),Succ_2(s)}(ren^{in}(Succ_2(s)),(\mathcal{V}(ren^{in}(s))\cup\mathcal{V}(ren^{out}(Succ_1(s),1)))\delta)\gamma'$$

to the other ones. Thus, we obtain k derivations $d_1,\ldots,d_k$ with $size(\pi)\leq 1+2\cdot size(\pi_1)\cdot size(\pi_2)\in\mathcal{O}(1+2\cdot cplx_{Succ_1(s)}(d_1^a,\ldots,d_k^a)\cdot cplx_{Succ_2(s)}(d_1^b,\ldots,d_k^b))=\mathcal{O}(cplx_{Succ_1(s)}(d_1^a,\ldots,d_k^a)\cdot cplx_{Succ_2(s)}(d_1^b,\ldots,d_k^b))\subseteq\mathcal{O}(cplx_s(d_1,\ldots,d_k))$.
• Otherwise, we consider the nodes in π from s to the first node $s'\in Inst(G)\cup Parallel(G)\cup Split(G)\cup Succ_1(Inst(G))$. If no such node s' exists, then π has only one branch and is contained in a subgraph of G without any INST nodes. By Lemma C 10, we know that then $size(\pi)\in\mathcal{O}(1)\subseteq\mathcal{O}(1+cplx_{root(G)}(d_1,\ldots,d_k))=\mathcal{O}(cplx_{root(G)}(d_1,\ldots,d_k))$. Thus, we consider the case that such a node s' exists. Then we directly have that the path from s to s' is a connection path π_c in G. Thus, we have the rule $ren^{in}(s)\sigma_{\pi_c,0}\to\mathsf{u}_{s,s'}(ren^{in}(s'),\mathcal{V}(ren^{in}(s)))\sigma_{\pi_c,0})$ in $\mathcal{R}(G)$. By Lemma C 7, we know that $\sigma_{\pi_c,0}$ is an instance of the answer substitution along π_c. In particular, we know that $ren^{in}(s)\sigma_{\pi_c,0}$ matches t by γ and we can rewrite t to $\mathsf{u}_{s,s'}(ren^{in}(s'),\mathcal{V}(ren^{in}(s)))\sigma_{\pi_c,0})\gamma$. We use the induction hypothesis for π' which is the subtree from s' of π and the full subgraph G' at s' to obtain that the term $ren^{in}(s')\gamma$ starts k derivation $d'_1,\ldots,d'_k$ with $size(\pi')\in\mathcal{O}(cplx_{root(G')}(d'_1,\ldots,d'_k))$. Together with the first rewrite step, we obtain $size(\pi)=|\pi_c|-1+size(\pi')\in\mathcal{O}(|\pi_c|-1+cplx_{root(G')}(d'_1,\ldots,d'_k))=\mathcal{O}(cplx_{root(G')}(d'_1,\ldots,d'_k))\subseteq\mathcal{O}(cplx_{root(G)}(d_1,\ldots,d_k))$.

□

We can now state the central theorem of this paper where we prove that the complexity of the resulting relative complexity problem is an asymptotic upper bound for the comeplexity of the original Prolog program w.r.t. the specified set of queries.

Theorem C 13 (*Complexity Analysis for Prolog Programs*)
Let $\mathcal{P}$ be a Prolog program, $\mathsf{p}\in\Sigma$ be a predicate, m be a moding function, and G be a decomposable derivation graph built for $\mathcal{P}$ and the class of queries $\mathcal{Q}_m^{\mathsf{p}}$. Then we have $prc_{\mathcal{P},\mathcal{Q}_m^{\mathsf{p}}}(n)\in\mathcal{O}(cplx_{root(G)}(n))$.

Proof
Let $Q=\mathsf{p}(t_1,\ldots,t_k)\in\mathcal{Q}_m^{\mathsf{p}}$ with $|Q|_m\leq n$ such that the evaluation of Q takes ℓ steps and ℓ is maximal, i.e., $prc_{\mathcal{P},\mathcal{Q}_m^{\mathsf{p}}}(n)=\ell$. By the construction of derivation graphs, we know that $root(G)=s=\mathsf{p}(T_1,\ldots,T_k);(\mathcal{G},\mathcal{F},\mathcal{U})$, there is a γ with $Q=\mathsf{p}(T_1,\ldots,T_k)\gamma\in\mathcal{CON}(S;(\mathcal{G},\mathcal{F},\mathcal{U}))$ and $\mathcal{G}=\{T_i\mid m(\mathsf{p},i)=\mathbf{in}\}$. Thus, we know that $|ren^{in}(s)\gamma|=|Q|_m$. By Lemma C 4 we obtain a tree path π_b for the evaluation of Q with $\ell\in\mathcal{O}(size(\pi_b))$. If $\mathcal{R}(G)=\varnothing$, then there is no PARALLEL or SPLIT node in G and no connection path in G. In particular, we cannot reach any INST node from s. By Lemma C 10, we conclude that $\ell\in\mathcal{O}(1)\subseteq\mathcal{O}(1+cplx_s(n))=\mathcal{O}(cplx_s(n))$. Thus, we consider the case $\mathcal{R}(G)\neq\varnothing$. By Lemma C 12, we know that $ren^{in}(s)\gamma$ is a basic term w.r.t. $\mathcal{R}(G)$ and starts k derivations $d_1,\ldots,d_k$ with the rules from $\mathcal{R}(G)$ such that $size(\pi_b)\in\mathcal{O}(cplx_s(d_1,\ldots,d_k))\subseteq\mathcal{O}(cplx_s(n))$. Together, we obtain

$prc_{\mathcal{P},\mathcal{Q}^{\mathrm{p}}_m}(n) = \ell \in \mathcal{O}(size(\pi_b)) \subseteq \mathcal{O}(cplx_s(d_1,\ldots,d_k)) \subseteq \mathcal{O}(cplx_s(n))$ and, hence, our theorem. □

Appendix D Proving Correctness of the Determinacy Criterion

Finally, we have to prove our result on the determinacy criterion. We first show a simple lemma.

Lemma D 1 (*No Success without* SUC *Nodes*)
Let G be a derivation graph. Let s be a node in G which does not reach any SUC *node in G. Then for any concretization of s, its evaluation does not produce any answer substitutions.*

Proof
Let γ be a concretization w.r.t. $s = S; KB$. We perform the proof of the lemma by contradiction, assuming that from $S\gamma$ in ℓ steps we first reach a node where the SUC rule is applied, and by induction over the lexicographic combination of first the length ℓ of the evaluation of $S\gamma$ and second the edge relation of G restricted to INST, PARALLEL, and SPLIT edges. Note that this induction relation is indeed well founded as the sum of the number of terms and goals in a state is strictly decreased by applying the rules PARALLEL or SPLIT while it stays equal by applying the INST rule. So every infinite sequence of rule applications with only these rules must eventually only use the INST rule. This is in contradiction to the requirement that derivation graphs do not contain cycles consisting of INST edges only.

For $\ell = 0$, we trivially have that the number of answer substitutions produced for $S\gamma$ is 0.

For $\ell > 0$, we perform a case analysis on s:

- If $s \in \mathit{Inst}(G)$, then we know that there is a variable renaming ρ such that $S\gamma\rho \in \mathcal{CON}(\mathit{Succ}_1(s))$. As the number of answer substitutions is independent of variable renamings, we can use the induction hypothesis directly to obtain the lemma.
- If $s \in \mathit{Parallel}(G)$, then we either have that the evaluation of $S\gamma$ reaches a concrete state in $\mathcal{CON}(\mathit{Succ}_2(s))$ or not. If not, then we know by Lemma A 9 that all goals in $\mathit{Succ}_2(s)$ (possibly except for the last one if it is a scope marker) must be dropped due to a cut. Then there is a concrete state $S' \in \mathcal{CON}(\mathit{Succ}_1(s))$ such that its evaluation is identical to the one of $S\gamma$ (possibly except for the last FAILURE step which might be missing for S'). Thus, we can use the induction hypothesis for $\mathit{Succ}_1(s)$ (we have traversed a PARALLEL edge) and directly obtain that the evaluation produces 0 answer substitutions. If the evaluation of $S\gamma$ does reach a concrete state in $\mathcal{CON}(\mathit{Succ}_2(s))$, however, we obtain by the induction hypothesis that the evaluations of the corresponding backtracking goals in the two successors of s do not produce any answer substitutions (we can apply the induction hypothesis as the evaluation parts for the two successors of s are both shorter than the complete evaluation). Thus, the evaluation of $S\gamma$ produces $0 + 0 = 0$ answer substitutions.
- If $s \in \mathit{Split}(G)$, then we have $S = t, Q$ and we know that the evaluation of $t\gamma$ is at most as long as the one of $S\gamma$. Thus, we can use the induction hypothesis on the first successor of s (there we have one less SPLIT edge to traverse) and obtain that no answer substitution is produced for $t\gamma$. It follows immediately that then $S\gamma$ can

also produce no answer substitutions (since Q is only called if $t\gamma$ succeeds, which is not the case).

• Otherwise, we applied a rule to s where the next concrete state of the evaluation is represented by one successor of s. We obtain the lemma by the induction hypothesis.

□

The determinacy criterion must be extended in the presence of PARALLEL rules.

Definition D 2 (*Determinacy Criterion*)
A node s in G satisfies the determinacy criterion *if condition (a) or (b) holds:*

(a) *All successors of s satisfy the determinacy criterion. Moreover, if $s \in Suc(G)$, then there is no non-empty path from s to some* SUC *node in G. Furthermore, if $s \in Parallel(G)$, then at most one successor of s can reach some* SUC *node in G and if some* SUC *node is reachable from s, then there is no non-empty path from s to itself in G.*
(b) *The node s is a* SPLIT *node and at least one of $Succ_1(s)$ or $Succ_2(s)$ cannot reach a* SUC *node in G.*

Note that this definition in fact corresponds to Def. 14 when no PARALLEL nodes exist in G. Now we are ready to prove the correctness of the determinacy criterion.

Theorem D 3 (*Soundness of Determinacy Criterion*)
Let G be a derivation graph. Let s be a node in G which satisfies the (extended) determinacy criterion of Def. D 2. Then for any concretization of s, its evaluation results in at most one answer substitution. Thus if s' is a SPLIT *node and $Succ_1(s')$ satisfies the determinacy criterion, then s' is not multiplicative.*

Proof
Let γ be a concretization w.r.t. $s = S; KB$. We perform the proof of the first part of the theorem by contradiction, assuming that from $S\gamma$ in ℓ steps for the second time we reach a node where the SUC rule is applied, and by induction over the lexicographic combination of first the length ℓ of the evaluation of $S\gamma$ and second the edge relation of G restricted to INST, PARALLEL, and SPLIT edges. Note that this induction relation is indeed well founded as the sum of the number of terms and goals in a state is strictly decreased by applying the rules PARALLEL or SPLIT while it stays equal by applying the INST rule. So every infinite sequence of rule applications with only these rules must eventually only use the INST rule. This is in contradiction to the requirement that derivation graphs do not contain cycles consisting of INST edges only.

For $\ell = 0$, we trivially have that the number of answer substitutions produced for $S\gamma$ is $0 \leq 1$.

For $\ell > 0$, we perform a case analysis on s:

• If $s \in Inst(G)$, then we know that there is a variable renaming ρ such that $S\gamma\rho \in \mathcal{CON}(Succ_1(s))$. As the number of answer substitutions is independent of variable renamings, we can use the induction hypothesis directly to obtain the first part of the theorem.

• If $s \in \mathit{Suc}(G)$, then we know that the evaluation must start with one application of the SUC rule resulting in a concrete state S' and producing one answer substitution for this step. Moreover, we know that $S' \in \mathcal{CON}(\mathit{Succ}_1(s))$. By Def. D 2 we know that $\mathit{Succ}_1(s)$ cannot reach any SUC node. Thus, we obtain by Lemma D 1 that the evaluation of S' does not produce any additional answer substitutions and, hence, it follows that the evaluation of $S\gamma$ produces exactly one answer substitution.
• If $s \in \mathit{Parallel}(G)$, then we either have that the evaluation of $S\gamma$ does reach a concrete state in $\mathcal{CON}(\mathit{Succ}_2(s))$ or not. If not, then we know by Lemma A 9 that all goals in $\mathit{Succ}_2(s)$ (possibly except for the last one if it is a scope marker) must be dropped due to a cut. Then there is a concrete state $S' \in \mathcal{CON}(\mathit{Succ}_1(s))$ such that its evaluation is identical to the one of $S\gamma$ (possibly except for the last FAILURE step which might be missing for S'). Thus, we can use the induction hypothesis for $\mathit{Succ}_1(s)$ and directly obtain that the evaluation produces at most one answer substitution. If the evaluation of $S\gamma$ reaches a concrete state in $\mathcal{CON}(\mathit{Succ}_2(s))$, however, we can use the knowledge that s cannot reach itself or it cannot reach any SUC node. If s cannot reach any SUC node, so do both successors of s and we obtain by Lemma D 1 that the evaluations of the corresponding backtracking goals do not produce any answer substitutions. Thus, the evaluation of $S\gamma$ produces $0+0 = 0$ answer substitutions. If s cannot reach itself, $\mathit{Succ}_1(s)$ and $\mathit{Succ}_2(s)$ cannot reach s, either. Moreover, we know that only one successor of s can reach a SUC node. W.l.o.g. let $\mathit{Succ}_1(s)$ reach a SUC node (the other case is symmetric). By the induction hypothesis, we obtain that the evaluation of the backtracking goals in $\mathit{Succ}_1(s)$ produce at most one answer substitution while we know by Lemma D 1 that the backtracking goals in $\mathit{Succ}_2(s)$ produce no answer substitution. So the total number of answer substitutions for $S\gamma$ is at most 1.
• If $s \in \mathit{Split}(G)$, then we have $S = t, Q$ and we know that the evaluation of $t\gamma$ is at most as long as the one of $S\gamma$. Thus, we can use the induction hypothesis on the first successor of s and obtain that at most one answer substitution δ is produced for $t\gamma$. Hence, in case the evaluation reaches a state $Q\gamma\delta$, this state is represented by $\mathit{Succ}_2(s)$ as we know by Lemma A 10. The induction hypothesis is therefore applicable to $\mathit{Succ}_2(s)$ as well and we obtain that the evaluation of $Q\gamma\delta$ also produces at most one answer substitution. Together, the evaluation of $S\gamma$ produces at most one answer substitution as the numbers of solutions for $t\gamma$ and $Q\gamma\delta$ have to be multiplied. If already $t\gamma$ does not produce any solutions, so does $S\gamma$ and the first part of the theorem holds. If s cannot reach any SUC node, we know by Lemma D 1 that $S\gamma$ produces no answer substitutions.
• Otherwise, we applied a rule to s where the next concrete state of the evaluation is represented by one successor of s and no answer substitution is produced in that step. We obtain the first part of the theorem by the induction hypothesis.

For the last part of the theorem, we now know that $\mathit{Succ}_1(s')$ produces at most one answer substitution, i.e., the number of answer substitutions for the first successor of s' is bounded by a constant. Hence, s' is not multiplicative. □

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

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

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2009-01 *	Fachgruppe Informatik: Jahresbericht 2009
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
2009-05	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

2010-01 * Fachgruppe Informatik: Jahresbericht 2010

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

2010-05 Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme

2010-06 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata

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

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

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

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

2010-11 Martin Zimmermann: Parametric LTL Games

2010-12 Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut

2010-13 Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems

2010-14 Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination

2010-15 Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode

2010-16 Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles

2010-17 Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten

2010-18 Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit

2010-19 Felix Reidl: Experimental Evaluation of an Independent Set Algorithm

2010-20 Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games

2011-01 * Fachgruppe Informatik: Jahresbericht 2011

2011-02 Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting

2011-03 Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems

2011-04 Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars

2011-07 Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV

2011-08	Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog
2011-09	Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing
2011-10	Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations
2011-11	Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains
2011-12	Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems
2011-13	Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP
2011-14	Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games
2011-16	Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM
2011-18	Kamal Barakat: Introducing Timers to pi-Calculus
2011-19	Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode
2011-24	Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations
2011-25	Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations
2011-26	Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata
2012-01 *	Fachgruppe Informatik: Annual Report 2012
2012-02	Thomas Heer: Controlling Development Processes
2012-03	Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems
2012-04	Marcus Gelderie Strategy Machines and their Complexity

* These reports are only available as a printed version.

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