TLP **10** (4–6): 365–381, 2010. © Cambridge University Press 2010
doi:10.1017/S1471068410000165

*Automated termination analysis for logic programs with cut**

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

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

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

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

submitted 7 February 2010; revised 9 April 2010; accepted 14 May 2010

Abstract

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

KEYWORDS: automated termination analysis, cut, definite logic programs

1 Introduction

Automated termination analysis for logic programs has been widely studied (see, e.g. De Schreye and Decorte 1994; Codish *et al.* 2005; Serebrenik and De Schreye 2005; Bruynooghe *et al.* 2007; Mesnard and Serebrenik 2007; Schneider-Kamp *et al.* 2009; Nguyen *et al.* 2010.) Still, virtually all existing techniques only prove universal termination of *definite* logic programs, which do not use the *cut* "!". An exception is Marchiori (1996), which transforms "safely typed" logic programs to term rewrite systems (TRSs). However, the resulting TRSs are complex and since there is no implementation of Marchiori (1996), it is unclear whether they can be handled by existing TRS termination tools. Moreover, Marchiori (1996)'s method does not allow arbitrary cuts (e.g., it does not operate on programs like Example 1).

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

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

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

Example 1

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

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

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

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

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

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

2 Concrete derivations

See e.g. Apt (1997) for the basics of logic programming. We distinguish between individual cuts to make their scope explicit. So a signature Σ contains all predicate

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

and function symbols and all labeled versions of the cut $\{!_m/0 \mid m \in \mathbb{N}\}$. For simplicity we just consider terms $\mathscr{T}(\Sigma, \mathscr{V})$ and no atoms, i.e., we do not distinguish between predicate and function symbols. To ease the presentation, in the paper we exclude terms with cuts $!_m$ as proper subterms. A *clause* is a pair $H \leftarrow B$ where the *head* H is from $\mathscr{T}(\Sigma, \mathscr{V})$ and the *body* B is a sequence of terms from $\mathscr{T}(\Sigma, \mathscr{V})$. Let $Goal(\Sigma, \mathscr{V})$ be the set of all such sequences, where $\square$ is the empty goal.

A *program* $\mathscr{P}$ (possibly with cut) is a finite sequence of clauses. $Slice(\mathscr{P}, t)$ are all clauses for t's predicate, i.e., $Slice(\mathscr{P}, p(t_1, ..., t_n)) = \{c \mid c = \text{“}p(s_1, ..., s_n) \leftarrow B\text{”} \in \mathscr{P}\}$.

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

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

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

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

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

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

Definition 2 (*Semantics with Concrete Inference Rules*)

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

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

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

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

$$\frac{(t,Q)^i_m \mid S}{B'_i\sigma, Q\sigma \mid S}\ (\textsc{Eval})$$

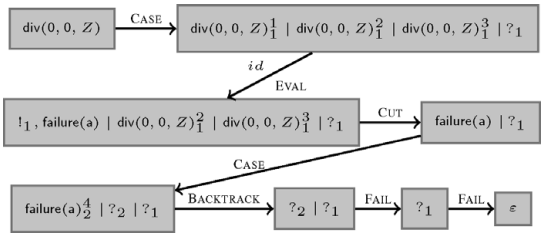
where $c_i = H_i \leftarrow B_i$, $mgu(t, H_i) = \sigma$, $B'_i = B_i[! \,/\, !_m]$.

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

where $c_i = H_i \leftarrow B_i$ and $t \not\sim H_i$.

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

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



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

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

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

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

the additional PARALLEL inference rule which will be introduced in Section 4. We treat such states as if $?_m$ were added at the end of the backtracking sequence.

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

The rules of Definition 2 define the semantics of logic programs with cut using states. They can also be used to define the semantics using derivations between goals: there is a derivation from the goal Q to Q' in the program $\mathscr{P}$ (denoted $Q \vdash^*_{\mathscr{P},\theta} Q'$) iff repeated application of our rules can transform the state[1] Q to a state of the form $\overline{Q'} \mid S$ for some S, and Q' results from $\overline{Q'}$ by removing all labels. Moreover, $\theta = \theta_1\theta_2 \ldots \theta_n$ where $\theta_1, \ldots, \theta_n$ are the mgu's used in those applications of the EVAL rule that led to $\overline{Q'}$. We call $\theta_{|\mathscr{V}(Q)}$ the corresponding *answer substitution*. If θ is not of interest, we write $\vdash_{\mathscr{P}}$ instead of $\vdash_{\mathscr{P},\theta}$.

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

3 Abstract derivations

To represent *classes* of queries, we introduce *abstract terms* and a set $\mathscr{A}$ of *abstract variables*, where each $T \in \mathscr{A}$ represents a fixed but arbitrary term. $\mathscr{N}$ consists of all "ordinary" variables in logic programming. Then, as *abstract terms* we consider all terms from the set $\mathscr{T}(\Sigma, \mathscr{V})$ where $\mathscr{V} = \mathscr{N} \uplus \mathscr{A}$. *Concrete terms* are terms from $\mathscr{T}(\Sigma, \mathscr{N})$, i.e., terms containing no abstract variables. For any set $\mathscr{V}' \subseteq \mathscr{V}$, let $\mathscr{V}'(t)$ be the variables from $\mathscr{V}'$ occurring in the term t.

To determine by which terms an abstract variable may be instantiated, we add a knowledge base $KB = (\mathscr{G}, \mathscr{U})$ to each state, where $\mathscr{G} \subseteq \mathscr{A}$ and $\mathscr{U} \subseteq \mathscr{T}(\Sigma, \mathscr{V}) \times \mathscr{T}(\Sigma, \mathscr{V})$. The variables in $\mathscr{G}$ may only be instantiated by ground terms. And $(s, s') \in \mathscr{U}$ means that we are restricted to instantiations γ of the abstract variables where $s\gamma \not\asymp s'\gamma$, i.e., s and s' may not become unifiable when instantiating them with γ.

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

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

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

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

Definition 4 (*Concretization*)
A substitution γ is a *concretization* w.r.t. $(\mathscr{G}, \mathscr{U})$ iff $Dom(\gamma) = \mathscr{A}$, $\mathscr{V}(Range(\gamma)) \subseteq \mathscr{N}$, $\mathscr{V}(Range(\gamma_{|\mathscr{G}})) = \emptyset$, and $s\gamma \not\sim s'\gamma$ for all $(s, s') \in \mathscr{U}$. The set of concretizations of an abstract state $(S; KB)$ is $\mathscr{C}on(S; KB) = \{S\gamma \mid \gamma \text{ is a concretization w.r.t. } KB\}$.

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

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

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

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

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

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

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

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

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

In Definition 2, we determined which of the rules Eval and Backtrack to apply by trying to unify the first atom t with the head H_i of the corresponding clause. But as demonstrated by Example 2, in the abstract case we might need to apply Eval for some concretizations and Backtrack for others. Backtrack can be used for *all* concretizations if t does not unify with H_i or if their mgu contradicts $\mathscr{U}$. This gives rise to the abstract Backtrack rule in the following definition. When the abstract

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

BACKTRACK rule is not applicable, we still cannot be sure that $t\gamma$ unifies with H_i for all concretizations γ. Thus, we have an abstract EVAL rule with two successor states that combines both the concrete EVAL and the concrete BACKTRACK rule.

Definition 7 (*Abstract Inference Rules – Part 2* (BACKTRACK, EVAL))

$$\frac{(t, Q)^i_m \mid S; KB}{S; KB}\ (\text{BACKTRACK}) \quad \text{where } c_i = H_i \leftarrow B_i \text{ and there is no concretization } \gamma \text{ w.r.t. } KB \text{ such that } t\gamma \sim H_i.$$

$$\frac{(t, Q)^i_m \mid S; (\mathscr{G}, \mathscr{U})}{B'_i\sigma, Q\sigma \mid S\sigma_{|\mathscr{G}}; (\mathscr{G}', \mathscr{U}\sigma_{|\mathscr{G}}) \qquad S; (\mathscr{G}, \mathscr{U} \cup \{(t, H_i)\})}\ (\text{EVAL})$$

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

In EVAL, w.l.o.g. we assume that $mgu(t, H_i)$ renames all variables to fresh abstract variables. This is needed to handle "sharing" effects correctly, i.e., to handle concretizations which introduce multiple occurrences of (concrete) variables, cf. Schneider-Kamp *et al.* (2010). The knowledge base is updated differently for the successors corresponding to the concrete EVAL and BACKTRACK rule. For all concretizations corresponding to the second successor of EVAL, the concretization of t does not unify with H_i. Hence, here we add the pair (t, H_i) to the set $\mathscr{U}$.

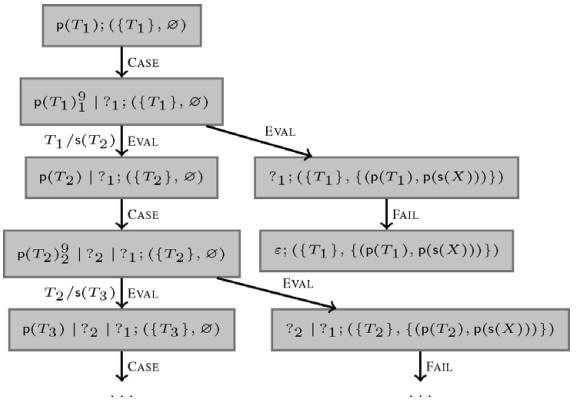
Now consider concretizations γ where $t\gamma$ and H_i unify, i.e., concretizations γ corresponding to the first successor of the EVAL rule. Then for any $T \in \mathscr{G}$, $T\gamma$ is a ground instance of $T\sigma$. Hence, we replace all $T \in \mathscr{G}$ by $T\sigma$, i.e., we apply $\sigma_{|\mathscr{G}}$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

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

Example 3
Consider the 1-rule program

$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X). \qquad (9)$$

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



4 From trees to graphs

To obtain a finite graph instead of an infinite tree, we now introduce an additional INSTANCE rule which allows us to connect the current state $(S; KB)$ with a previous

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

state $(S';KB')$, provided that the current state is an instance of the previous state. In other words, every concretization of $(S;KB)$ must be a concretization of $(S';KB')$. Still, INSTANCE is often not enough to obtain a finite graph.

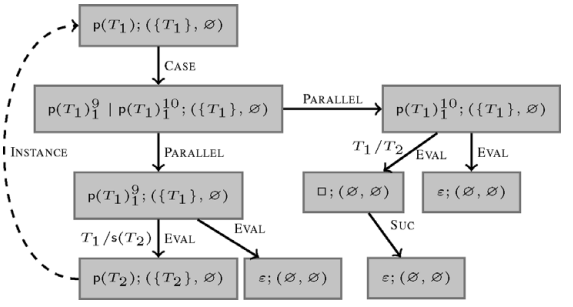
Example 4
We extend Example 3 by the following additional fact.

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

For queries $\mathsf{p}(t)$ where t is ground, the program still terminates. If we start with $(\mathsf{p}(T_1);(\{T_1\},\varnothing))$, then the CASE rule results in the state $(\mathsf{p}(T_1)^9_1 \mid \mathsf{p}(T_1)^{10}_1 \mid ?_1;(\{T_1\},\varnothing))$ and the EVAL rule produces two new states, one of them being $(\mathsf{p}(T_2) \mid \mathsf{p}(\mathsf{s}(T_2))^{10}_1 \mid ?_1;(\{T_2\},\varnothing))$.

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

We will also reduce the knowledge base to just those abstract variables that occur in the state and remove pairs (s,s') from $\mathscr{U}$ where $s \not\asymp s'$. Still, $(\mathsf{p}(T_2) \mid \mathsf{p}(\mathsf{s}(T_2))^{10}_1;(\{T_2\},\varnothing))$ is not an instance of the previous state $(\mathsf{p}(T_1);(\{T_1\},\varnothing))$ due to the added backtrack goal $\mathsf{p}(\mathsf{s}(T_2))^{10}_1$. Therefore, we now introduce a PARALLEL rule that allows us to split a backtracking sequence into separate problems. Now we obtain the graph on the right.



Clearly, PARALLEL may transform terminating into non-terminating states. But without further conditions, PARALLEL is not only "incomplete", but also unsound. Consider a state $(\,!_2 \mid !_1 \mid ?_2 \mid \mathsf{p};(\varnothing,\varnothing))$ for the program $\mathsf{p} \leftarrow \mathsf{p}$. The state is not terminating, as $!_1$ is not reachable. Thus, one eventually evaluates p. But if one splits the state into $(!_2;(\varnothing,\varnothing))$ and $(!_1 \mid ?_2 \mid \mathsf{p};(\varnothing,\varnothing))$, both new states terminate.

To solve this problem, in addition to the "active marks" (cf. Example 4) we introduce the notion of *active cuts*. The active cuts of a state S are those $m \in \mathbb{N}$ where $!_m$ occurs in S or where $!_m$ can be introduced by EVAL applied to a labeled goal $(t,q)^i_m$ occurring in S. Now the PARALLEL rule may only split a backtracking sequence into two parts S and S' if the active cuts of S and the active marks of S' are disjoint.

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

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

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

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

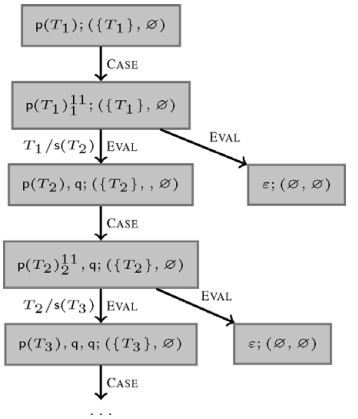
The *active cuts* $AC(S)$ are all m where $!_m$ is in S or $(t,q)^i_m$ is in S and c_i's body has a cut. The *active marks* $AM(S)$ are all m where $S = S' \mid ?_m \mid S''$ and $S' \neq \varepsilon, S'' \neq \varepsilon$.

Example 5

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

$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X), \mathsf{q}. \quad (11) \qquad \mathsf{q}. \quad (12)$$

For queries $\mathsf{p}(t)$ where t is ground, the program again terminates. With Definition 6, 7, and 8, we obtain the infinite tree on the right. It never encounters an instance of a previous state, since each resolution with Clause (11) adds a q to the goal.



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

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

$$\frac{t, Q; (\mathscr{G}, \mathscr{U})}{t; (\mathscr{G}, \mathscr{U}) \qquad Q\mu; (\mathscr{G}', \mathscr{U}\mu)}\ (\text{SPLIT})$$

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

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

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

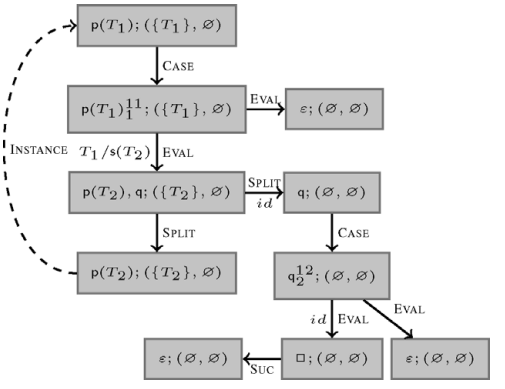
Example 6

To illustrate Definition 9, regard the program of Example 1 and the state $(\mathsf{sub}(T_5, T_6, T_8), \mathsf{div}(T_8, T_6, T_7); (\{T_5, T_6\}, \mathscr{U}))$ with $T_5, T_6, T_7, T_8 \in \mathscr{A}$. (This state will occur in the termination proof of div, cf. Example 7.) We have $\mathscr{G} = \{T_5, T_6\}$ and hence if $\mathsf{sub}(t_1, t_2, t_3)$ is $\mathsf{sub}(T_5, T_6, T_8)$, then $\mathit{Ground}_{\mathscr{P}}(\mathsf{sub}, \{i \mid \mathscr{V}(t_i) \subseteq \mathscr{G}\}) = \mathit{Ground}_{\mathscr{P}}(\mathsf{sub}, \{1, 2\}) = \{1, 2, 3\}$. In other words, if the first two arguments of sub are ground and the derivation is successful, then the answer substitution also instantiates the third argument to a ground term. Since μ only renames variables

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

outside of $\mathscr{G}$, we have $\mu = \{T_7/T_9, T_8/T_{10}\}$. So $\mathit{ApproxGnd}(\mathsf{sub}(T_5, T_6, T_8), \mu) = \{\mathscr{A}(t_1\mu), \mathscr{A}(t_2\mu), \mathscr{A}(t_3\mu)\} = \{T_5\mu, T_6\mu, T_8\mu\} = \{T_5, T_6, T_{10}\}$. So the SPLIT rule transforms the current state to $(\mathsf{sub}(T_5, T_6, T_8); (\{T_5, T_6\}, \mathscr{U}))$ and $(\mathsf{div}(T_{10}, T_6, T_9); (\{T_5, T_6, T_{10}\}, \mathscr{U}\mu))$ where one can eliminate T_5 from the new groundness set $\mathscr{G}'$.

With the additional SPLIT rule, we can always obtain finite graphs instead of infinite trees. (This will be proved in Theorem 2.) Thus, no further rules are needed. As depicted on the right, now we can also close the graph for Example 5's program.



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

Theorem 1 (*Soundness of the Abstract Inference Rules*)
The inference rules from Definition 6, 7, 8, and 9 are sound[2].

5 From termination graphs to logic programs

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

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

We want to generate clauses for the loops in the termination graph and show their termination. Thus, there should be no cycles consisting only of INSTANCE edges, as they would lead to trivially non-terminating clauses. Moreover, the only leaves may be nodes where no inference rule is applicable anymore (i.e., the graph must be "fully expanded"). For example, the graph at the end of Section 4 is a termination graph. Theorem 2 shows that termination graphs can always be obtained automatically.

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

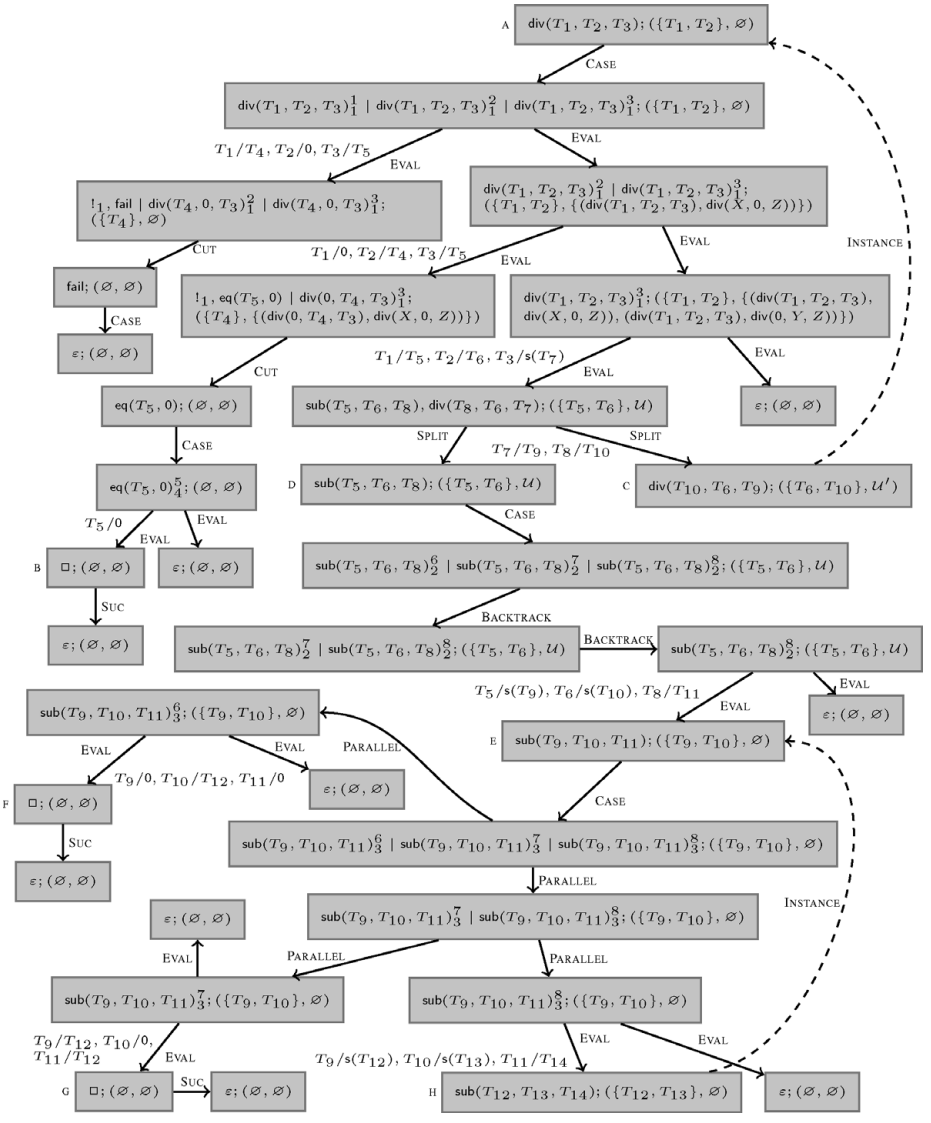
Example 7
For the program from Example 1 we obtain the termination graph below. Here, $\mathscr{U} = \{(\mathsf{div}(T_5, T_6, T_3), \mathsf{div}(X, \mathsf{0}, Z)), (\mathsf{div}(T_5, T_6, T_3), \mathsf{div}(\mathsf{0}, Y, Z))\}$ results from exploiting the cuts. $\mathscr{U}$ implies that neither T_6 nor T_5 unify with 0. Thus, only Clause (8) is applicable to evaluate the state in Node D. This is crucial for termination, because

[2] For all proofs, we refer to Schneider-Kamp *et al.* (2010).

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

in D, sub's result T_8 is always smaller than sub's input argument T_5 and therefore, div's first argument in Node C is smaller than div's first argument in Node A.

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



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

More precisely, we build clauses for all *clause paths*. For a termination graph G, let INSTANCE(G) denote all nodes of G to which the rule INSTANCE has been applied

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

(i.e., C and H in our example). The sets SPLIT(G) and SUC(G) are defined analogously. For any node n, let $Succ(i, n)$ denote the i-th child of n. Clause paths are paths in the graph that start in the root node, in the successor node of an INSTANCE node, or in the left child of a SPLIT node and that end in a SUC or INSTANCE node or in the left child of an INSTANCE or SPLIT node.

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

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

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

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

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

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

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

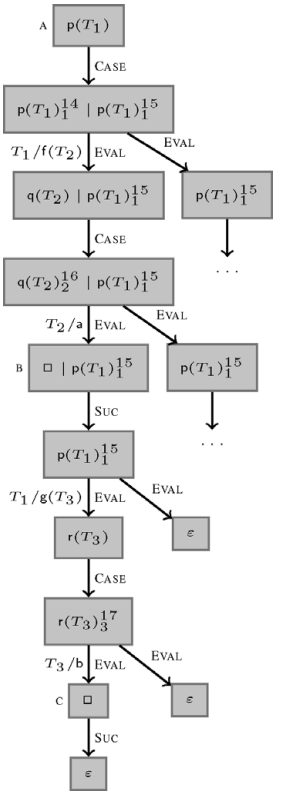
$$
\begin{aligned}
\mathsf{div}_{\text{A}}(0, T_4, 0). & \\
\mathsf{div}_{\text{A}}(T_5, T_6, \mathsf{s}(T_9)) &\leftarrow \mathsf{sub}_{\text{D}}(T_5, T_6, T_{10}), \mathsf{div}_{\text{A}}(T_{10}, T_6, T_9). \qquad (13)\\
\mathsf{div}_{\text{A}}(T_5, T_6, \mathsf{s}(T_7)) &\leftarrow \mathsf{sub}_{\text{D}}(T_5,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

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

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

In general, to convert a node n into an atom, we use a function *Ren*. *Ren*(n) has the form $p_n(X_1, \ldots, X_n)$ where p_n is a fresh predicate symbol for the node n (except if n is an INSTANCE node) and $X_1, \ldots, X_n$ are all variables in n. This renaming allows us to use different predicate symbols for different nodes. For example, the cut-free logic program above would not terminate if we identified sub_D and sub_E. The reason is that sub_D only succeeds if its first and second argument start with "s". Hence, if the intermediate body atom $\mathsf{sub}_\mathrm{D}(T_5, T_6, T_{10})$ of Clause (13) succeeds, then the "number T_{10}" will always be strictly smaller than the "number T_5". Finally, *Ren* allows us to represent a whole state by just one atom, even if this state consists of a non-atomic goal or a backtracking sequence with several elements.

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



Example 5

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

$$\mathsf{p}(\mathsf{f}(X)) \leftarrow \mathsf{q}(X). \quad (14) \qquad \mathsf{q}(\mathsf{a}). \quad (16)$$

$$\mathsf{p}(\mathsf{g}(X)) \leftarrow \mathsf{r}(X). \quad (15) \qquad \mathsf{r}(\mathsf{b}). \quad (17)$$

This graph contains clause paths from A to B and from A to C. For every clause path, we collect the relevant substitutions step by step, starting from the end of the path. So for the first clause path we start with $\{T_2/\mathsf{a}\}$. This substitution results from an EVAL node for the goal $\mathsf{q}(T_2)_2^{16}$ with mark $d = 2$. Hence, for the first clause path we only collect further substitutions that result from EVAL nodes with marks smaller than $d = 2$. Since the next substitution $\{T_1/\mathsf{f}(T_2)\}$ results from an EVAL node with mark 1, we finally obtain $\{T_1/\mathsf{f}(T_2)\} \circ \{T_2/\mathsf{a}\}$ which leads to the fact $\mathsf{p}(\mathsf{f}(\mathsf{a}))$ in the resulting logic program. For the second clause path from A to C, we start with $\{T_3/\mathsf{b}\}$ which results from an EVAL node with mark $d = 3$. When moving upwards in the tree, the substitution $\{T_1/\mathsf{g}(T_3)\}$ also has to be collected,

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

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

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

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

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

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

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

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

So if n_{j-1} is a SPLIT node, then one has to "collect" the corresponding substitution μ when constructing the overall substitution $\sigma_{n_1 \dots n_j, d}$ for the path. If n_{j-1} is an EVAL node for the m-th case analysis and n_j is its left successor, then the construction of $\sigma_{n_1 \dots n_j, d}$ depends on whether we have already collected a corresponding substitution for the current case analysis m. If m is smaller than the mark d for the last case analysis which contributed to the substitution, then the corresponding substitution σ of the EVAL rule is collected and d is set to m. Otherwise (if $d \leqslant m$), one only collects the part $\sigma_{|\mathscr{G}}$ of the substitution that concerns those abstract variables that stand for ground terms. The definition of the intermediate body atoms I_π ensures

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

that derivations in $\mathscr{P}_G$ only reach the second child of a SPLIT node if the first child of the SPLIT node could successfully be proved.

Theorem 3 proves the soundness of our approach. So termination of the cut-free program $\mathscr{P}_G$ implies termination of the original program $\mathscr{P}$. (However as shown in Schneider-Kamp *et al.* (2010), the converse does not hold.)

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

6 Experiments and conclusions

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

We implemented this pre-processing in our tool AProVE Giesl *et al.* (2006) and performed extensive experiments which show that now we can indeed prove termination of typical logic programs with cut fully automatically. The implementation is not only successful for programs like Example 1, but also for programs using operators like *negation as failure* or *if then else* which can be expressed using cuts. While AProVE was already very powerful for termination analysis of definite logic programs Schneider-Kamp *et al.* (2009), our pre-processing method strictly increases its power. For our experiments, we used the *Termination Problem Database* (TPDB) of the annual *International Termination Competition*[3]. Since up to now, no tool had special support for cuts, the previous versions of the TPDB did not contain any programs with cuts. Therefore, we took existing cut-free examples from the TPDB and added cuts in a natural way. In this way, we extended the TPDB by 104 typical programs with cuts (directory `LP/CUT`). Of these, 10 are known to be non-terminating. Up to now, termination tools treated cuts by simply ignoring them and by trying to prove termination of the program that results from removing the cuts. This is sensible, since cuts are not always needed for termination. Indeed, a version of AProVE that ignores cuts and does not use our pre-processing can show termination of 10 of the 94 potentially terminating examples. Other existing termination tools would not yield much better results, since AProVE is already the most powerful tool for definite logic programming (as shown by the experiments in Schneider-Kamp *et al.* (2009)) and since most of the remaining 84 examples do not terminate anymore if one removes the cut. In contrast, with our new pre-processing, AProVE proves termination of 78 examples (i.e., 83% of the potentially terminating examples). This shows that our contributions are crucial for termination analysis of logic programs with cuts. Nevertheless, there is of course room for further improvements (e.g., one could develop alternative techniques to generate cut-free clauses from the

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

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

termination graph in order to improve the performance on examples which encode existential termination). To experiment with our implementation and for further details, we refer to http://aprove.informatik.rwth-aachen.de/eval/Cut/.

Acknowledgements

We thank the referees for many helpful remarks.

References

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

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

Billaud, M. 1990. Simple operational and denotational semantics for Prolog with cut. *Theoretical Computer Science 71,* 2, 193–208.

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

Codish, M., Lagoon, V., and Stuckey, P. J. 2005. Testing for termination with monotonicity constraints. In Logic Programming, 21st International Conference, ICLP, M. Gabbrielli and G. Gupta, Eds. *'05*. Lecture Notes in Computer Science, vol. 3668. Springer, Berlin, 326–340.

Cousot, P. and Cousot, R. 1992. Abstract interpretation and application to logic programs. *Journal of Logic Programming 13,* 2–3, 103–179.

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

de Vink, E. P. 1989. Comparative semantics for Prolog with cut. *Science of Computer Programming 13,* 1, 237–264.

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

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

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, vol. 4130. Springer, Berlin, 281–286.

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

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

Kulas, M. and Beierle, C. 2000. Defining standard Prolog in rewriting logic. In *WRLA '00*. Electronic Notes in Theoretical Computer Science, vol. 36. Elsevier, the Netherlands, 39–59.

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

Marchiori, M. 1996. Proving existential termination of normal logic programs. In *AMAST '96*. Lecture Notes in Computer Science, vol. 1101. Springer, Berlin, 375–390.

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

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press

Mogensen, T. Æ. 1996. A semantics-based determinacy analysis for Prolog with cut. In *Ershov Memorial Conference*. Lecture Notes in Computer Science, vol. 1181. Springer, Berlin, 374–385.

Nguyen, M. T., De Schreye, D., Giesl, J., and Schneider-Kamp, P. 2010. Polytool: Polynomial interpretations as a basis for termination analysis of logic programs. *Theory and Practice of Logic Programming* (to appear).

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., Serebrenik, A., Ströder, T., and Thiemann, R. 2010. Automated termination analysis for logic programs with cut. Tech. Rep. AIB-2010-10, RWTH Aachen. http://aib.informatik.rwth-aachen.de.

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

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

Spoto, F. 2000. Operational and goal-independent denotational semantics for Prolog with cut. *Journal of Logic Programming 42,* 1, 1–46.

Spoto, F. and Levi, G. 1998. Abstract interpretation of Prolog programs. In *AMAST '98*. Lecture Notes in Computer Science, vol. 1548. Springer, Berlin, 455–470.

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

https://doi.org/10.1017/S1471068410000165 Published online by Cambridge University Press