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. 2011) 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 $\mathsf{f}(t_1, \dots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$ for all $\mathsf{f}/n \in \Sigma$ and $t_1, \dots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$. $\mathcal{V}(t)$ denotes the set resp. the sequence of variables in a term t. For a term $t = \mathsf{f}(t_1, \dots, t_n)$, 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 $t = \mathsf{f}(t_1, \dots, t_n)$, we have $t|_{pos} = t|_{i\ pos'} = t_i|_{pos'}$ and $t[s]_{pos} = t[s]_{i\ pos'} = \mathsf{f}(t_1, \dots, t_i[s]_{pos'}, \dots, t_n)$.

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 $s \overset{i}{\to}_\mathcal{R} t$) 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 $\mathsf{f}(t_1, \dots, t_n)$, where $\mathsf{f} \in \Sigma_d$ and $t_1, \dots, t_n$ 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 $\overset{i}{\to}_\mathcal{R}$-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 $\Box$ 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 :- $\Box$". 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 $\langle G_1 \mid \dots \mid G_n \rangle$ where $G_1 \mid \dots \mid G_n$ 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 $(t_1, \dots, t_k)^c$ indicates that the next resolution step has to be performed with clause c. Intuitively, a state $\langle G_1 \mid \dots \mid G_n \rangle$ means that we currently have to solve the goal G_1, but that $G_2, \dots, G_n$ are the next goals to solve when backtracking.[5] The *initial state* for a query $(t_1, \dots, t_k)$ is $\langle (t_1, \dots, t_k) \rangle$, 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 $\mathcal{Q}_m^{\mathsf{p}} = \{\mathsf{p}(t_1,\ldots,t_n) \mid \mathcal{V}(t_i) = \varnothing \text{ for all } i \text{ with } m(\mathsf{p},i) = in\}$. 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 $\mathcal{Q}_m^{\mathsf{p}}$, the Prolog *runtime complexity function* $prc_{\mathcal{P},\mathcal{Q}_m^{\mathsf{p}}}$ maps any $n \in \mathbb{N}$ to the length of the longest derivation starting with the initial state for some query $Q \in \mathcal{Q}_m^{\mathsf{p}}$ with $|Q|_m \le n$. For a program $\mathcal{P}$ and a class of queries $\mathcal{Q}_m^{\mathsf{p}}$, our aim is to generate a TRS $\mathcal{R}$ such that asymptotically, $irc_{\mathcal{R}}(n)$ is an upper bound of $prc_{\mathcal{P},\mathcal{Q}_m^{\mathsf{p}}}(n)$.

3 Direct Transformation

Consider the following program `sublist.pl` from the *Termination Problem Data Base (TPDB)*[7] with the class of queries $\mathcal{Q}_m^{\mathsf{sublist}}$. 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 p^{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 $\mathsf{p}^{in}(\vec{s}) \to \mathsf{p}^{out}(\vec{t})$.
- For each clause c of the form $\mathsf{p}(\vec{s},\vec{t}) \,\text{:-}\, \mathsf{p}_1(\vec{s_1},\vec{t_1}),\dots,\mathsf{p}_k(\vec{s_k},\vec{t_k})$, the resulting TRS contains the following rules:

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

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.

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

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.

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

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

$$\frac{(t,Q)^{h \,\text{:-}\, B} \mid S \;;\; \mathcal{G}}{(B\sigma, Q\sigma) \mid S\sigma|_{\mathcal{G}} \;;\; \mathcal{G}' \qquad S \;;\; \mathcal{G}} \text{ (EVAL)}$$

where $mgu(t,h) = \sigma$. 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 $\mathcal{G}' = \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}}))$.

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

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 $mgu(t,h) = \sigma$ 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 $\sigma = mgu(\mathsf{app}(T_{11}, T_8, \overline{T_9}), \mathsf{app}([\,], Ys, Ys)) = \{T_8/\overline{T_{12}}, \overline{T_9}/\overline{T_{12}}, T_{11}/[\,], Ys/\overline{T_{12}}\}$ which leads to $(\langle \square \mid \mathsf{app}(T_{11}, T_8, \overline{T_{12}})^{(2)} \rangle \;;\; \{\overline{T_{12}}\})$. 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.

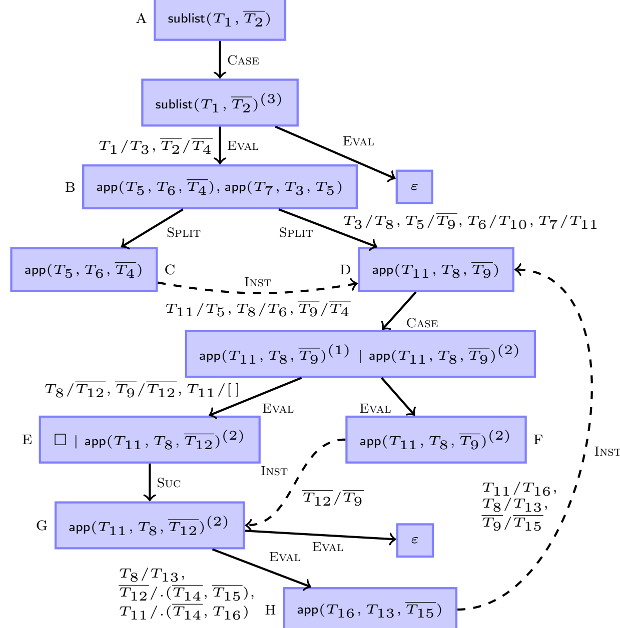

Fig. 3. Derivation Graph for the `sublist` Program

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 $(\mathsf{app}(T_5, T_6, \overline{T_4}), \mathsf{app}(T_7, T_3, T_5))$ in state B. Otherwise, when evaluating the first atom $\mathsf{app}(T_5, T_6, \overline{T_4})$ 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 $\mathsf{app}(T_{14}, T_{15}, \overline{T_{13}}), \mathsf{app}(T_{16}, T_{10}, .(\overline{T_{12}}, T_{14}))$. 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 $\mathsf{app}(T_5, T_6, \overline{T_4})$ 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 (Ströder et al. 2012) 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 $\mathit{NextG}(\mathsf{app}(T_5, T_6, \overline{T_4}), \mathcal{G})\delta = \{\delta(T_5), \delta(T_6)\} = \{\overline{T_9}, \overline{T_{10}}\}$. 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 $\mathit{Rule}(G)$ denote all nodes of G to which Rule has been applied. We denote by $\mathit{Succ}_i(s)$ the i-th child of node s and by $\mathit{Succ}_i(\mathit{Rule}(G))$ the set of i-th children of all nodes from $\mathit{Rule}(G)$.

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 $\mathit{ren}^{in}(\text{B}) = f_{\text{B}}^{in}(T_4)$ (since $\mathcal{G} = \{T_4\}$ in B) and $\mathit{ren}^{out}(\text{B}) = f_{\text{B}}^{out}(T_5, T_6, T_7, T_3)$ (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 $ren^{in}(\textsc{c}) = ren^{in}(\textsc{d})\,\mu = f^{in}_{\textsc{d}}(T_9)\,\mu = f^{in}_{\textsc{d}}(T_4)$ and $ren^{out}(\textsc{c}) = ren^{out}(\textsc{d})\,\mu = f^{out}_{\textsc{d}}(T_{11}, T_8)\,\mu = f^{out}_{\textsc{d}}(T_5, T_6)$.

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

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

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

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

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

So to compute $NextG((t_1,\dots,t_k),\mathcal{G})$ for a query $(t_1,\dots,t_k)$, 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 $\pi = s_1 \dots s_k$ 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. 2011), 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 (Ströder et al. 2012) 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 $\mathsf{sublist}(T_3, \overline{T_4}) \mathrel{:-} \mathsf{app}(T_5, T_6, \overline{T_4}), \mathsf{app}(T_7, T_3, T_5)$, 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

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

However, connection paths π' like the one from D to E where the end node is a SUC node, are considered like a clause $\mathsf{app}([\,], \overline{T_{12}}, \overline{T_{12}}) \mathrel{:-} \Box$, 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

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

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 (6a) and (6b) when evaluating a term built with f^{in}_{D}.

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

Definition 6 (*Rules for Connection Paths*)
For a connection path $\pi = s_1 \ldots s_k$, 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):

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

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,

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

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.

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

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

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

Definition 7 (*Rules for Split Nodes, Corresponding TRS of a Derivation Graph*)
Let $s \in Split(G)$, $s_1 = Succ_1(s)$, and $s_2 = Succ_2(s)$. Moreover, let δ be the substitution associated to s. Then $SplitRules(s) =$

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

So the TRS $\mathcal{R}(G)$ corresponding *to G consists of $ConnectionRules(\pi)$ 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 $\stackrel{i}{\to}_{\mathcal{R}'}$-steps that occur in any sequence of $\stackrel{i}{\to}_{\mathcal{R}}$-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.

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 $irc_{\mathcal{R}_{A}(G),\mathcal{R}(G)}(n) + irc_{\mathcal{R}_{B}(G),\mathcal{R}(G)}(n) \cdot$

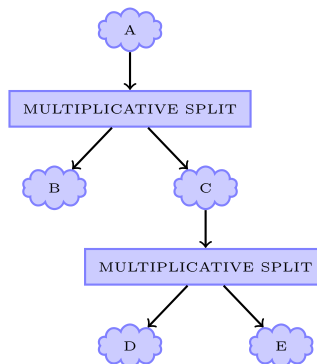

Fig. 8. Decomposing Graphs

$(irc_{\mathcal{R}_C(G),\mathcal{R}(G)}(n) + irc_{\mathcal{R}_D(G),\mathcal{R}(G)}(n) \cdot irc_{\mathcal{R}_E(G),\mathcal{R}(G)}(n))$ 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*(π) for the connection path π from A to B and of *SplitRules*(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. 12 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 (Ströder et al. 2012).

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 $\mathcal{Q}_m^{\mathsf{p}}$. Then $prc_{\mathcal{P},\mathcal{Q}_m^{\mathsf{p}}}(n) \in \mathcal{O}(cplx_{root(G)}(n))$.

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 $prc_{\mathcal{P},\mathcal{Q}_m^{\mathsf{sublist}}}(n) \in \mathcal{O}(n^2)$.

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/`.[14]

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.

[14] This website also contains an extended version of the paper with all proofs (Ströder et al. 2012).

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., 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., Schneider-Kamp, P., and Giesl, J. 2011. 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/`.

Ströder, T., Emmes, F., Giesl, J., Schneider-Kamp, P., and Fuhs, C. 2012. Automated complexity analysis for Prolog by term rewriting. Tech. Rep. AIB 2012-05, RWTH Aachen. Available from `http://aib.informatik.rwth-aachen.de/` and 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.