Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs *

Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Germany
giesl@informatik.rwth-aachen.de

Thomas Ströder
LuFG Informatik 2, RWTH Aachen University, Germany
stroeder@informatik.rwth-aachen.de

Peter Schneider-Kamp
Dept. of Mathematics and Computer Science, University of Southern Denmark
petersk@imada.sdu.dk

Fabian Emmes
LuFG Informatik 2, RWTH Aachen University, Germany
emmes@informatik.rwth-aachen.de

Carsten Fuhs
Dept. of Computer Science, University College London, United Kingdom
c.fuhs@cs.ucl.ac.uk

Abstract

There exist many powerful techniques to analyze *termination* and *complexity* of *term rewrite systems* (TRSs). Our goal is to use these techniques for the analysis of other programming languages as well. For instance, approaches to prove termination of definite logic programs by a transformation to TRSs have been studied for decades. However, a challenge is to handle languages with more complex evaluation strategies (such as Prolog, where predicates like the *cut* influence the control flow). In this paper, we present a general methodology for the analysis of such programs. Here, the logic program is first transformed into a *symbolic evaluation graph* which represents all possible evaluations in a finite way. Afterwards, different analyses can be performed on these graphs. In particular, one can generate TRSs from such graphs and apply existing tools for termination or complexity analysis of TRSs to infer information on the termination or complexity of the original logic program.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical Verification; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Automatic Analysis of Algorithms

General Terms Languages, Theory, Verification

Keywords Logic Programs, Prolog, Term Rewriting, Termination, Complexity, Determinacy

* 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.

1. Introduction

We are concerned with analyzing "semantical" properties of logic programs, like *termination*, *complexity*, and *determinacy* (i.e., the question whether all queries in a specific class succeed at most once). While there are techniques and tools that analyze logic programs *directly*, we present a general *transformational* methodology for such analyses. In this way, one can re-use existing powerful techniques and tools that have been developed for *term rewriting*.

For well-moded definite logic programs, there are several transformations to TRSs such that termination of the TRS implies termination of the original logic program [33]. We extended these transformations to arbitrary definite programs in [35].

However, Prolog programs typically use the *cut* predicate. To handle the non-trivial control flow induced by cuts, in [37] we introduced a pre-processing method where a Prolog program is first transformed into a *symbolic evaluation graph*. (These graphs were inspired by related approaches to program optimization [38] and were called "termination graphs" in [37].) Symbolic evaluation graphs also represent those aspects of the program that cannot easily be expressed in term rewriting. We also developed similar approaches for other programming languages like Java and Haskell [7–9, 17]. For Prolog, the transformation from the program to the symbolic evaluation graph relies on a new "linear" operational semantics which we presented in [41]. From the symbolic evaluation graph, one can then generate a simpler program (without cuts) whose termination implies termination of the original Prolog program. In [37] we generated definite logic programs from the graph (whose termination could then be analyzed by transforming them further to TRSs, for example). In [40], we presented a more powerful approach which generates so-called *dependency triples* [31, 36] from the graph.

In the current paper, we show that the symbolic evaluation graph cannot only be used for termination analysis, but it is also very suitable as the basis for several other analyses, such as complexity or determinacy analysis. So symbolic evaluation graphs and term rewriting can be seen as a general methodology for the analysis of programming languages like Prolog.[1]

[1] This methodology can also be used to analyze programs in other languages. For example, in [8] we used similar graphs not just for termination proofs, but also for disproving termination and for detecting `NullPointerExceptions` in Java programs.

After recapitulating the underlying operational semantics in Sect. 2, we introduce the symbolic evaluation graph in Sect. 3. To use this graph for different forms of program analysis, we present several new theorems which express the connection between the "abstract evaluations" represented in the graph and the "concrete evaluations" of actual queries.

In Sect. 4, we present a new improved approach for termination analysis of logic programs, where one directly generates *term rewrite systems* from the symbolic evaluation graph. This results in a substantially more powerful approach than [37]. Compared to [40], our new approach is considerably simpler and it allows us to apply *any* tool for termination of TRSs when analyzing the termination of logic programs. So one does not need tools that handle the (non-standard) notion of "dependency triples" anymore.

In Sect. 5 we show that symbolic evaluation graphs and the TRSs generated from the graphs can also be used in order to analyze the *complexity* of logic programs. Here, we rely on recent results which show how to adapt techniques for termination analysis of TRSs in order to prove asymptotic upper bounds for the runtime complexity of TRSs automatically.

Finally, Sect. 6 demonstrates that the symbolic evaluation graph can also be used to analyze whether a class of queries is *deterministic*. Besides being interesting on its own, such a determinacy analysis is also needed in our new approach for complexity analysis of logic programs in Sect. 5.

We implemented all our contributions in our automated termination tool AProVE [15] and performed extensive experiments to compare our approaches with existing analysis techniques which work directly on logic programs. It turned out that our approaches for termination and complexity clearly outperform related existing techniques. For determinacy analysis, our approach can handle many examples where existing methods fail, but there are also many examples where the existing techniques are superior. Thus, here it would be promising to couple our approach with existing ones. All proofs can be found in [18].

2. Preliminaries and Operational Semantics of Prolog

See, e.g., [2] for the basics of logic programming. We label individual cuts to make their scope explicit. Thus, we use a signature Σ containing $\{!_m/0 \mid m \in \mathbb{N}\}$ and all predicate and function symbols. As in the ISO standard for Prolog [23], we do not distinguish between predicate and function symbols and just consider *terms* $\mathcal{T}(\Sigma, \mathcal{V})$ and no atoms.

A *query* is a sequence of terms. Let $Query(\Sigma, \mathcal{V})$ denote the set of all queries, where $\square$ is 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 we write just "h" instead of "h :- $\square$". A *logic program* $\mathcal{P}$ is a finite sequence of clauses.

We now briefly recapitulate our operational semantics from [41], which is equivalent to the ISO semantics in [23]. As shown in [41], both semantics yield the same answer substitutions, the same termination behavior, and the same complexity. The advantage of our semantics is that it is particularly suitable for an extension to *classes* of queries, i.e., for the symbolic evaluation of *abstract* states, cf. Sect. 3. This makes our semantics particularly well suited for analyzing logic programs.

Our semantics is given by a set of inference rules that operate on *states*. A *state* has the form $(G_1 \mid \ldots \mid G_n)$ where each G_i is a *goal*. Here, G_1 represents the current query and $(G_2 \mid \ldots \mid G_n)$ represents the queries that have to be considered next. This backtrack information is contained in the state in order to describe the effect of cuts. Since each state contains all backtracking goals, our semantics is *linear* (i.e., an *evaluation* with these rules is just a sequence of states and not a search tree as in the ISO semantics).

Essentially, a *goal* is just a query, i.e., a sequence of terms. But to compute answer substitutions, a goal is labeled by a substitution which collects the unifiers used up to now. So if $(t_1, \ldots, t_k)$ is a query, then a goal has the form $(t_1, \ldots, t_k)_\theta$ for a substitution θ. In addition, a goal can also be labeled by a clause c, where $(t_1, \ldots, t_k)^c_\theta$ means that the next resolution has to be performed with clause c. Moreover, a goal can also be a *scope marker* $?_m$ for $m \in \mathbb{N}$. This marker denotes the end of the scope of cuts $!_m$ labeled with m. Whenever a cut $!_m$ is reached, all goals preceding $?_m$ are discarded.

Def. 1 shows the inference rules for the part of Prolog defining definite logic programming and the cut. See [41] for the inference rules for full Prolog. Here, S and S' are states and the query Q may also be $\square$ (then "(t, Q)" is t).

DEFINITION 1 (Operational Semantics).

$$\frac{\square_\theta \mid S}{S}\ (\textsc{Suc}) \qquad \frac{(t,Q)^{h\,:-\,B}_\theta \mid S}{(B\sigma, Q\sigma)_{\theta\sigma} \mid S}\ (\textsc{Eval})\ \textit{if } mgu(t,h) = \sigma$$

$$\frac{?_m \mid S}{S}\ (\textsc{Fail}) \qquad \frac{(t,Q)^{h\,:-\,B}_\theta \mid S}{S}\ (\textsc{Backtrack})\ \textit{if } t \not\sim h$$

$$\frac{(t,Q)_\theta \mid S}{(t,Q)^{c_1[!/!_m]}_\theta \mid \ldots \mid (t,Q)^{c_a[!/!_m]}_\theta \mid ?_m \mid S}\ (\textsc{Case})$$

where t is no cut or variable, m is fresh, and $Slice_\mathcal{P}(t) = (c_1, \ldots, c_a)$

$$\frac{(!_m, Q)_\theta \mid S \mid ?_m \mid S'}{Q_\theta \mid ?_m \mid S'}\ (\textsc{Cut})\ \textit{where } S \textit{ contains no } ?_m \qquad \frac{(!_m, Q)_\theta \mid S}{Q_\theta}\ (\textsc{Cut})\ \textit{where } S \textit{ contains no } ?_m$$

The SUC rule is applicable if the first goal of our sequence could be proved. Then we 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 evaluation 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 rule determines which clauses could be applied to t by slicing the program according to t's root symbol. Here, $Slice_\mathcal{P}(\mathsf{p}(t_1, \ldots, t_n))$ is the sequence of all program clauses "h :- B" from $\mathcal{P}$ where $root(h) = \mathsf{p}/n$. The variables in program clauses are renamed when this is necessary to ensure variable-disjointness with the states. Thus, CASE replaces the current goal $(t, Q)_\theta$ by a goal labeled with the first such clause and adds copies of $(t, Q)_\theta$ labeled by the other potentially applicable clauses as backtracking possibilities. Here, the top-down clause selection rule is taken into account. The cuts in these clauses are labeled by a fresh mark $m \in \mathbb{N}$ (i.e., $c[!/!_m]$ is the clause c where all cuts ! are replaced by $!_m$), and $?_m$ is added at the end of the new backtracking goals to denote their scope.

EXAMPLE 2. *Consider the following logic program.*

$$\mathsf{star}(XS, [\,]) \text{ :- } !. \quad (1)$$

$$\mathsf{star}([\,], ZS) \text{ :- } !, \mathsf{eq}(ZS, [\,]). \quad (2)$$

$$\mathsf{star}(XS, ZS) \text{ :- } \mathsf{app}(XS, YS, ZS), \mathsf{star}(XS, YS). \quad (3)$$

$$\mathsf{app}([\,], YS, YS). \quad (4)$$

$$\mathsf{app}([X \mid XS], YS, [X \mid ZS]) \text{ :- } \mathsf{app}(XS, YS, ZS). \quad (5)$$

$$\mathsf{eq}(X, X). \quad (6)$$

Here, $\mathsf{star}(t_1, t_2)$ *holds iff* t_2 *results from repeated concatenation of* t_1. *So we have* $\mathsf{star}([1,2],[\,])$, $\mathsf{star}([1,2],[1,2])$, $\mathsf{star}([1,2],[1,2,$

$1, 2]$*), etc. The cut in rule (2) is needed for termination of queries of the form* $\mathsf{star}([\,], t)$. *For the query* $\mathsf{star}([1, 2], [\,])$, *we obtain the following evaluation, where we omitted the labeling by substitutions for readability.*

$$\begin{array}{rl}
\mathsf{star}([1,2],[\,]) & \vdash_{\textsc{Case}} \\
\mathsf{star}([1,2],[\,])^{(1')} \mid \mathsf{star}([1,2],[\,])^{(2')} \mid \mathsf{star}([1,2],[\,])^{(3)} \mid ?_1 & \vdash_{\textsc{Eval}} \\
!_1 \mid \mathsf{star}([1,2],[\,])^{(2')} \mid \mathsf{star}([1,2],[\,])^{(3)} \mid ?_1 & \vdash_{\textsc{Cut}} \\
\square \mid ?_1 & \vdash_{\textsc{Suc}} \\
?_1 & \vdash_{\textsc{Fail}} \ \varepsilon
\end{array}$$

So the CASE *rule results in a state which represents a case analysis where we first try to apply the* star*-clause (1). The state also contains the next backtracking goals, since when backtracking later on, we would use clauses (2) and (3). Here,* $(1')$ *denotes* $(1)[!/!_1]$ *and* $(2')$ *denotes* $(2)[!/!_1]$.

For a goal $(t,Q)_\theta^{h \,:-\, B}$, if t unifies[2] with the head h of the program clause, we apply EVAL. This rule replaces t by the body B of the clause and applies the mgu σ to the result. Moreover, σ contributes to the answer substitution, i.e., we replace the label θ by $\theta\sigma$.

If t does not unify with h (denoted "$t \not\sim h$"), we apply the BACKTRACK rule. Then, $h \,:-\, B$ cannot be used and we backtrack to the next goal 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 its scope is explicitly represented by $!_m$ and $?_m$, we have turned the cut into a *local* operation depending only on the current state. Note that $?_m$ must not be deleted as the current goal Q_θ 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). We treat such states as if $?_m$ were added at the end of the state.

For each query Q, its corresponding *initial state* consists of just $(Q[!/!_1])_{id}$ (i.e., all cuts in Q are labeled by a fresh number like 1 and the goal is labeled by the identity substitution id). The query Q is *terminating* if all evaluations starting in its corresponding initial state are finite. Our inference rules can also be used to define *answer substitutions*.

DEFINITION 3 (Answer Substitution). *Let S be a state with a single goal Q_σ (which may additionally be labeled by a clause c). We say that θ is an* answer substitution *for S if there is an evaluation from S to a state $(\square_{\sigma\theta} \mid S_{\mathit{suffix}})$ for a (possibly empty) state S_{suffix} (i.e., $(\square_{\sigma\theta} \mid S_{\mathit{suffix}})$ is obtained by repeatedly applying rules from Def. 1 to S). Similarly, θ is an* answer substitution *for a query if it is an answer substitution for the query's initial state.*

3. From Prolog to Symbolic Evaluation Graphs

We now explain the construction of *symbolic evaluation graphs* which represent all evaluations of a logic program for a certain *class* of queries. While we already presented such graphs in [37], here we introduce a new formulation of the corresponding abstract inference rules which is suitable for generating TRSs afterwards. Moreover, we present new theorems (Thm. 5, 8, and 10) which express the exact connection between abstract and concrete evaluations. These theorems will be used to prove the soundness of our analyses later on.

We consider classes of atomic queries described by a $\mathsf{p}/n \in \Sigma$ and a *moding function* $m : \Sigma \times \mathbb{N} \to \{in, out\}$. So m determines which arguments of a symbol are "inputs". The corresponding class of queries is $\mathcal{Q}_m^{\mathsf{p}} = \{\mathsf{p}(t_1, \ldots, t_n) \mid \mathcal{V}(t_i) = \varnothing$ for all i with $m(\mathsf{p}, i) = in\}$. Here, "$\mathcal{V}(t_i)$" denotes the set of all variables occurring in t_i. So for the program of Ex. 2, we might regard the class of queries $\mathcal{Q}_m^{\mathsf{star}}$ where $m(\mathsf{star}, 1) = m(\mathsf{star}, 2) = in$. Thus, $\mathcal{Q}_m^{\mathsf{star}} = \{\mathsf{star}(t_1, t_2) \mid t_1, t_2$ are ground$\}$.

To represent classes of queries, we regard *abstract* states that stand for sets of concrete states. Instead of "ordinary" variables $\mathcal{N}$, abstract states use abstract variables $\mathcal{A} = \{T_1, T_2, \ldots\}$ representing fixed, but arbitrary terms (i.e., $\mathcal{V} = \mathcal{N} \uplus \mathcal{A}$).

To obtain concrete states from an abstract one, we use *concretizations*. A concretization is a substitution γ which replaces all abstract variables by concrete terms, i.e., $Dom(\gamma) = \mathcal{A}$ and $\mathcal{V}(Range(\gamma)) \subseteq \mathcal{N}$. To determine by which terms an abstract variable may be instantiated, we add a knowledge base $KB = (\mathcal{G}, \mathcal{U})$ to each state, where $\mathcal{G} \subseteq \mathcal{A}$ and $\mathcal{U} \subseteq \mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{T}(\Sigma, \mathcal{V})$. The variables in $\mathcal{G}$ may only be instantiated by ground terms, i.e., $\mathcal{V}(Range(\gamma|_\mathcal{G})) = \varnothing$. Here, "$\gamma|_\mathcal{G}$" denotes the restriction of γ to $\mathcal{G}$, i.e., $\gamma|_\mathcal{G}(X) = \gamma(X)$ for $X \in \mathcal{G}$ and $\gamma|_\mathcal{G}(X) = X$ for $X \in \mathcal{V} \setminus \mathcal{G}$. A pair $(t, t') \in \mathcal{U}$ means that we are restricted to concretizations γ where $t\gamma \not\sim t'\gamma$, i.e., t and t' must not be unifiable after γ is applied. Then we say that γ is a concretization *w.r.t.* KB.

Thus, an abstract state has the form $(S; KB)$. Here, S has the form $(G_1 \mid \ldots \mid G_n)$ where the G_i are goals over the signature Σ and the abstract variables $\mathcal{A}$ (i.e., they do not contain variables from $\mathcal{N}$). In contrast to [37], we again label all goals (except scope markers) by substitutions $\theta : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{A})$ in order to store which substitutions were applied during an evaluation. These substitution labels will be necessary for the synthesis of TRSs in Sect. 4.

The notion of *concretization* can also be used for states. A (concrete) state S' is a concretization of $(S; KB)$ if there exists a concretization γ w.r.t. KB such that S' results from $S\gamma$ by replacing the substitution labels of its goals by arbitrary (possibly different) substitutions $\theta : \mathcal{N} \to \mathcal{T}(\Sigma, \mathcal{N})$. To ease readability, we often write "$S\gamma$" to denote an arbitrary concretization of $(S; KB)$. Let $\mathcal{CON}(S; KB)$ denote the set of all concretizations of an abstract state $(S; KB)$.

For a class $\mathcal{Q}_m^{\mathsf{p}}$ with p/n, now the *initial state* is $(\mathsf{p}(T_1, \ldots, T_n)_{id}, (\mathcal{G}, \varnothing))$, where $\mathcal{G}$ contains all T_i with $m(\mathsf{p}, i) = in$.

We now adapt the inference rules of Def. 1 to abstract states. The rules SUC, FAIL, CUT, and CASE do not change the knowledge base and are straightforward to adapt. In Def. 1, we determined which of the rules EVAL and BACKTRACK to apply by trying to unify the first term t with the head h of the corresponding clause. But in the abstract case we might need to apply EVAL for some concretizations and BACKTRACK for others. The abstract BACKTRACK rule in Def. 4 can be used if $t\gamma$ does not unify with h for *any* concretization γ. Otherwise, $t\gamma$ unifies with h for *some* concretizations γ, but possibly not for *others*. Thus, the abstract EVAL rule has two successor states to combine both the concrete EVAL and the concrete BACKTRACK rule. Consequently, we now obtain symbolic evaluation trees instead of sequences.

DEFINITION 4 (Abstract Inference Rules).

$$\frac{(\square_\theta \mid S); KB}{S; KB}\ (\textsc{Suc}) \qquad \frac{((!_m, Q)_\theta \mid S \mid ?_m \mid S'); KB}{(Q_\theta \mid ?_m \mid S'); KB}\ (\textsc{Cut})\ \textit{where } S \textit{ contains no } ?_m$$

$$\frac{(?_m \mid S); KB}{S; KB}\ (\textsc{Fail}) \qquad \frac{((!_m, Q)_\theta \mid S); KB}{Q_\theta; KB}\ (\textsc{Cut})\ \textit{where } S \textit{ contains no } ?_m$$

$$\frac{((t, Q)_\theta \mid S); KB}{((t, Q)_\theta^{c_1[!/!_m]} \mid \ldots \mid (t, Q)_\theta^{c_a[!/!_m]} \mid ?_m \mid S); KB}\ (\textsc{Case})$$

where t is no cut or variable, m is fresh, $Slice_\mathcal{P}(t) = (c_1, \ldots, c_a)$

[2] In this paper, we consider unification with occurs check. Our method could be extended to unification without occurs check, but we left this as future work since most programs do not rely on the absence or presence of the occurs check.

$$\frac{((t,Q)_\theta^{h\,:\text{-}\,B} \mid S); KB}{S; KB} \text{ (BACKTRACK)} \quad \textit{if there is no concretization } \gamma \textit{ w.r.t. } KB \textit{ such that } t\gamma \sim h.$$

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

if $mgu(t,h) = \sigma$. *W.l.o.g.,* $\mathcal{V}(Range(\sigma))$ *only contains fresh abstract variables and* $Dom(\sigma)$ *contains all previously occurring variables. Moreover,* $\mathcal{G}' = \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ *and* S' *results from* S *by applying the substitution* $\sigma|_{\mathcal{G}}$ *to its goals and by composing* $\sigma|_{\mathcal{G}}$ *with the substitution labels of its goals.*

To handle "sharing" effects correctly [37], w.l.o.g. we assume that $mgu(t,h) = \sigma$ renames all occurring variables to fresh abstract variables in EVAL. 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. Hence, here we add (t,h) to $\mathcal{U}$.

Now consider concretizations γ where $t\gamma$ and h unify, i.e., these concretizations γ correspond to the first successor of the EVAL rule. Then for any $T \in \mathcal{G}$, $T\gamma$ is a ground instance of $T\sigma$. Hence, we replace all $T \in \mathcal{G}$ by $T\sigma$, i.e., we apply $\sigma|_{\mathcal{G}}$ to S. The new set $\mathcal{G}'$ of variables that may only be instantiated by ground terms are the abstract variables occurring in $Range(\sigma|_{\mathcal{G}})$ (denoted "$\mathcal{A}(Range(\sigma|_{\mathcal{G}}))$"). As before, t is replaced by the instantiated clause body B and the previous substitution label θ is composed with the mgu σ (yielding $\theta\sigma$).

Thm. 5 states that any concrete evaluation with Def. 1 can also be simulated with the abstract rules of Def. 4.

THEOREM 5 (Soundness of Abstract Rules). *Let* $(S; KB)$ *be an abstract state with a concretization* $S\gamma \in \mathcal{CON}(S; KB)$*, and let* S_{next} *be the successor of* $S\gamma$ *according to the operational semantics in Def. 1. Then the abstract state* $(S; KB)$ *has a successor* $(S'; KB')$ *according to an inference rule from Def. 4 such that* $S_{next} \in \mathcal{CON}(S'; KB')$.

As an example, consider the program from Ex. 2 and the class of queries $\mathcal{Q}_m^{\mathsf{star}}$. The corresponding initial state is $(\mathsf{star}(T_1,T_2)_{id}; (\{T_1,T_2\},\varnothing))$. A *symbolic evaluation* starting with this state A is depicted in Fig. 6. The nodes of such a symbolic evaluation graph are states and each step from a node to its children is done by an inference rule. To save space, we omitted the knowledge base from the states $(S; (\mathcal{G},\mathcal{U}))$. Instead, we overlined all variables contained in $\mathcal{G}$ and labeled those edges where new information is added to $\mathcal{U}$.

The child of A is B with $(\mathsf{star}(\overline{T_1},\overline{T_2})_{id}^{(1')} \mid \mathsf{star}(\overline{T_1},\overline{T_2})_{id}^{(2')} \mid \mathsf{star}(\overline{T_1},\overline{T_2})_{id}^{(3)} \mid ?_1)$. In Fig. 6 we simplified the states by removing markers $?_m$ that occur at the end of a state. This is possible, since applying the first CUT rule to a state ending in $?_m$ corresponds to applying the second CUT rule to the same state without $?_m$. Moreover, $(1')$ and $(2')$ again abbreviate $(1)[!/!_1]$ and $(2)[!/!_1]$.

In B, $(1')$ is used for the next evaluation. EVAL yields two successors: In C, $\sigma_1 = mgu(\mathsf{star}(\overline{T_1},\overline{T_2}), \mathsf{star}(XS,[\,])) = \{\overline{T_1}/\overline{T_3}, XS/\overline{T_3}, \overline{T_2}/[\,]\}$ leads to $((!_1)_{\sigma_1} \mid \mathsf{star}(\overline{T_3},[\,])_{\sigma_2}^{(2')} \mid \mathsf{star}(\overline{T_3},[\,])_{\sigma_2}^{(3)})$. Here, $\sigma_2 = \sigma_1|_{\{\overline{T_1},\overline{T_2}\}}$. In the second successor D of B, we add the information $\mathsf{star}(\overline{T_1},\overline{T_2}) \not\sim \mathsf{star}(XS,[\,])$ to $\mathcal{U}$ (thus, we labeled the edge from B to D accordingly).

Unfortunately, even for terminating queries, in general the rules of Def. 4 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. To represent all possible evaluations in a finite way, we need additional inference rules to obtain finite symbolic evaluation graphs instead of infinite trees.

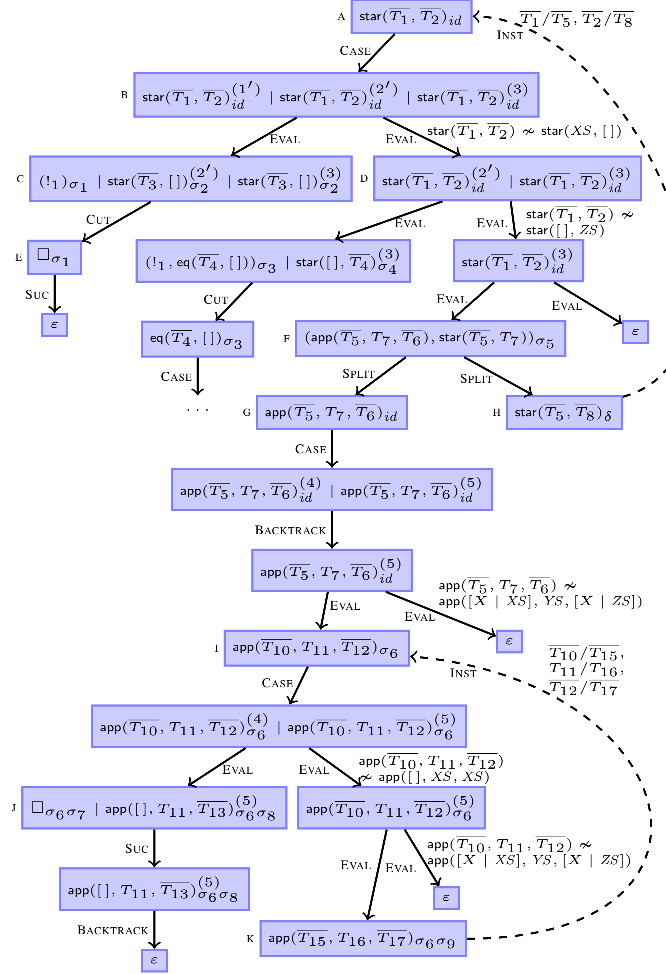


Figure 6. Symbolic Evaluation Graph for Ex. 2

To this end, we use an additional INST rule which allows us to connect the current state $(S; KB)$ with a previous state $(S'; KB')$, provided that the current state is an instance of the previous state. In other words, every concretization of $(S; KB)$ must be a concretization of $(S'; KB')$. More precisely, there must be a matching substitution μ such that $S'\mu = S$ up to the substitutions used for labeling goals in S' and S. These substitution labels do not have to be taken into account here, since we will not generate rewrite rules from paths that traverse INST edges in Sect. 4. Moreover, for $KB' = (\mathcal{G}', \mathcal{U}')$ and $KB = (\mathcal{G}, \mathcal{U})$, $\mathcal{G}'$ and $\mathcal{G}$ must be the same (modulo μ) and all constraints from $\mathcal{U}'$ must occur in $\mathcal{U}$ (modulo μ). Then we say that μ is *associated* to $(S; KB)$ and label the resulting INST edge with μ. For example, in Fig. 6, $\mu = \{\overline{T_1}/\overline{T_5}, \overline{T_2}/\overline{T_8}\}$ is associated to H and the edge from H to A is labeled with μ. We only define the INST rule for states containing a single goal. As indicated by our experiments, this is no severe restriction in practice.[3]

[3] In [37] and in our implementation, we use an additional inference rule to split up sequences of goals, but we omitted it here for readability. Adding this rule allows us to construct a symbolic evaluation graph for each program and query.

DEFINITION 7 (Abstract Rules: INST).

$$\frac{S;(\mathcal{G},\mathcal{U})}{S';(\mathcal{G}',\mathcal{U}')}\ (\text{INST})$$

if $S = Q_\theta$ and $S' = Q'_{\theta'}$, or $S = Q^c_\theta$ and $S' = Q'^c_{\theta'}$ for some non-empty queries Q and Q', such that there is a μ with $Dom(\mu) \subseteq \mathcal{A}$, $\mathcal{V}(Range(\mu)) \subseteq \mathcal{A}$, $Q = Q'\mu$, $\mathcal{G} = \bigcup_{T\in\mathcal{G}'} \mathcal{V}(T\mu)$, and $\mathcal{U}'\mu \subseteq \mathcal{U}$.

Thm. 8 states that every concrete state represented by an INST node is also represented by its successor.

THEOREM 8 (Soundness of INST). *Let $(S; KB)$ be an abstract state, let $(S'; KB')$ be its successor according to the* INST *rule, and let μ be associated to $(S; KB)$. If $S\gamma \in \mathcal{CON}(S;KB)$, then for $\gamma'=\mu\gamma$ we have $S'\gamma' \in \mathcal{CON}(S';KB')$.*

Moreover, we also need a SPLIT inference rule to split a state $((t, Q)_\theta; KB)$ into $(t_{id}; KB)$ and $((Q\delta)_\delta; KB')$, where δ approximates the answer substitutions for t. Such a SPLIT is often needed to make the INST rule applicable. We say that δ is *associated* to $((t, Q)_\theta; KB)$. The previous substitution label θ does not have to be taken into account here, since we will not generate rewrite rules from paths that traverse SPLIT nodes in Sect. 4. Thus, we can reset the substitution label θ to id in the first successor of the SPLIT node and store the associated substitution δ in the substitution label of the second successor. Similar to the INST rule, we only define the SPLIT rule for states containing a single goal.

DEFINITION 9 (Abstract Rules: SPLIT).

$$\frac{(t, Q)_\theta; (\mathcal{G},\mathcal{U})}{t_{id}; (\mathcal{G},\mathcal{U}) \quad (Q\delta)_\delta; (\mathcal{G}',\mathcal{U}\delta)}\ (\text{SPLIT})$$

where δ replaces all previously occurring variables from $\mathcal{A}\backslash\mathcal{G}$ by fresh abstract variables and $\mathcal{G}' = \mathcal{G} \cup NextG(t, \mathcal{G})\delta$.

Here, $NextG$ is defined as follows. We assume that we have a *groundness analysis* function $Ground_\mathcal{P} : \Sigma \times 2^\mathbb{N} \to 2^\mathbb{N}$, see, e.g., [22]. If $\mathsf{p}/n \in \Sigma$ and $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$, then $Ground_\mathcal{P}(\mathsf{p}, \{i_1, \ldots, i_m\}) = \{j_1, \ldots, j_k\}$ means that any query $\mathsf{p}(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{N})$ where $t_{i_1}, \ldots, t_{i_m}$ are ground only has answer substitutions θ where $t_{j_1}\theta, \ldots, t_{j_k}\theta$ are ground. So $Ground_\mathcal{P}$ approximates which positions of p will become ground if the "input" positions $i_1, \ldots, i_m$ are ground. Now if $t = \mathsf{p}(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{A})$ is an abstract term where $t_{i_1}, \ldots, t_{i_m}$ become ground in every concretization (i.e., all their variables are from $\mathcal{G}$), then $NextG(t, \mathcal{G})$ returns all variables in t that will be made ground by every answer substitution for any concretization of t. Thus, $NextG(t, \mathcal{G})$ contains the variables of $t_{j_1}, \ldots, t_{j_k}$. So formally

$$NextG(\mathsf{p}(t_1, \ldots, t_n), \mathcal{G}) = \bigcup_{j\in Ground_\mathcal{P}(\mathsf{p},\ \{i \mid \mathcal{V}(t_i)\subseteq\mathcal{G}\}\ )} \mathcal{V}(t_j).$$

Hence, in the second successor of the SPLIT rule, the variables in $NextG(t, \mathcal{G})$ can be added to the groundness set $\mathcal{G}$. Since these variables were renamed by δ, we extend $\mathcal{G}$ by $NextG(t, \mathcal{G})\delta$.

For instance, in Fig. 6, we split the query $\mathsf{app}(\overline{T_5}, T_7, \overline{T_6})$, $\mathsf{star}(\overline{T_5}, T_7)$ in state F. Thus, the first successor of F is $\mathsf{app}(\overline{T_5}, T_7, \overline{T_6})$ in state G. By groundness analysis, we infer that every successful evaluation of $\mathsf{app}(\overline{T_5}, T_7, \overline{T_6})$ instantiates T_7 by ground terms, i.e., $Ground_\mathcal{P}(\mathsf{app}, \{1, 3\}) = \{1, 2, 3\}$. Thus, for $\mathcal{G} = \{T_5, T_6\}$, we have $NextG(\mathsf{app}(T_5, T_7, T_6), \mathcal{G}) = \mathcal{V}(T_5)\cup\mathcal{V}(T_7)\cup\mathcal{V}(T_6) = \{T_5, T_7, T_6\}$. So in the second successor H of F, we use the substitution $\delta(T_7) = T_8$ and extend the groundness set $\mathcal{G}$ of F by $NextG(\mathsf{app}(T_5, T_7, T_6), \mathcal{G})\delta = \{T_5, T_8, T_6\}$. Thus, T_8 is also overlined in Fig. 6.

Thm. 10 shows the soundness of SPLIT. Suppose that we apply the SPLIT rule to $((t, Q)_\theta; KB)$, which yields $(t_{id}; KB)$ and $((Q\delta)_\delta; KB')$. Any evaluation of a concrete state $(t\gamma, Q\gamma) \in \mathcal{CON}((t, Q)_\theta; KB)$ consists of parts where one evaluates $t\gamma$ (yielding some answer substitution θ') and of parts where one evaluates $Q\gamma\theta'$. Clearly, those parts which correspond to evaluations of $t\gamma$ can be simulated by the left successor of the SPLIT node (since $t\gamma \in \mathcal{CON}(t_{id}; KB)$). Thm. 10 states that the parts of the overall evaluation which correspond to evaluations of $Q\gamma\theta'$ can be simulated by the right successor of the SPLIT node (i.e., $Q\gamma\theta' \in \mathcal{CON}((Q\delta)_\delta; KB')$).

THEOREM 10 (Soundness of SPLIT). *Let $((t,Q)_\theta;KB)$ be an abstract state and let $(t_{id}; KB)$ and $((Q\delta)_\delta; KB')$ be its successors according to the* SPLIT *rule. Let $(t\gamma, Q\gamma) \in \mathcal{CON}((t, Q)_\theta; KB)$ and let θ' be an answer substitution of $(t\gamma)_{id}$. Then we have $Q\gamma\theta' \in \mathcal{CON}((Q\delta)_\delta; KB')$.*

We define *symbolic evaluation graphs* as a subclass of the graphs obtained by the rules of Def. 4, 7, and 9. They must not have any cycles consisting only of INST edges, as this would lead to trivially non-terminating TRSs. Moreover, their only leaves may be nodes where no inference rule is applicable anymore (i.e., the graphs must be "fully expanded"). The graph in Fig. 6 is indeed a symbolic evaluation graph.

DEFINITION 11 (Symbolic Evaluation Graph). *A finite graph built from an initial state using Def. 4, 7, and 9 is a* symbolic evaluation graph *(or "evaluation graph" for short) iff there is no cycle consisting only of* INST *edges and all leaves are of the form $(\varepsilon; KB)$.*[4]

4. From Symbolic Evaluation Graphs to TRSs – Termination Analysis

Now our goal is to show termination of all concrete states represented by the graph's initial state. To this end, we synthesize a TRS from the symbolic evaluation graph. This TRS has the following property: if there is an evaluation from a concretization of one state to a concretization of another state which may be crucial for termination, then there is a corresponding rewrite sequence w.r.t. the TRS. Then automated tools for termination analysis of TRSs can be used to show termination of the synthesized TRS and this implies termination of the original logic program. See, e.g., [13, 16, 43] for an overview of techniques for automatically proving termination of TRSs.

For the basics of term rewriting, we refer to [6]. A *term rewrite system* $\mathcal{R}$ is a finite set of rules $\ell \to r$ where $\ell \notin \mathcal{V}$ and $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$. The rewrite relation $t \to_\mathcal{R} t'$ for two terms t and t' holds iff there is an $\ell \to r \in \mathcal{R}$, a position pos, and a substitution σ such that $\ell\sigma = t|_{pos}$ and $t' = t[r\sigma]_{pos}$. Here, $t|_{pos}$ is the subterm of t at position pos and $t[r\sigma]_{pos}$ results from replacing the subterm $t|_{pos}$ at position pos in t by the term $r\sigma$. The rewrite step is *innermost* (denoted $t \xrightarrow{i}_\mathcal{R} t'$) iff no proper subterm of $\ell\sigma$ can be rewritten.

To obtain a TRS from an evaluation graph Gr, we encode the states as terms. For each state $s = (S; (\mathcal{G},\mathcal{U}))$, we use two fresh function symbols f^{in}_s and f^{out}_s. The arguments of f^{in}_s are the variables in $\mathcal{G}$ (which represent ground terms). The arguments of f^{out}_s are those remaining abstract variables which will be made ground by every answer substitution for any concretization of s. They are again determined by groundness analysis [22]. Formally, the encoding of states is done by two functions enc^{in} and enc^{out}.

For instance, for the state F in Fig. 6, we obtain $enc^{in}(\text{F}) = f^{in}_\text{F}(T_5, T_6)$ (as $\mathcal{G} = \{T_5, T_6\}$ in F) and $enc^{out}(\text{F}) = f^{out}_\text{F}(T_7)$. The reason is that if γ instantiates T_5 and T_6 by ground terms, then

[4] The application of inference rules to abstract states is not deterministic. In our prover AProVE, we implemented a heuristic [39] to generate symbolic evaluation graphs automatically which turned out to be very suitable for subsequent analyses in our empirical evaluations.

every answer substitution of $(\mathsf{app}(T_5, T_7, T_6)\gamma, \mathsf{star}(T_5, T_7)\gamma)$ instantiates $T_7\gamma$ to a ground term as well.

For an INST node like H with associated substitution μ we do not introduce fresh function symbols, but use the function symbol of its (more general) successor instead. So we take the terms resulting from its successor A and apply μ to them. In other words, $enc^{in}(\text{H}) = enc^{in}(\text{A})\,\mu = f^{in}_{\text{A}}(T_1, T_2)\,\mu = f^{in}_{\text{A}}(T_5, T_8)$ and $enc^{out}(\text{H}) = enc^{out}(\text{A})\,\mu = f^{out}_{\text{A}}\,\mu = f^{out}_{\text{A}}$.

In the following, for an evaluation graph Gr and an inference rule RULE, $Rule(Gr)$ denotes all nodes of Gr to which RULE was applied. Let $Succ_i(s)$ denote the i-th child of node s and $Succ_i(Rule(Gr))$ denotes the set of i-th children of all nodes from $Rule(Gr)$.

DEFINITION 12 (Encoding States as Terms). *Let s be an abstract state with a single goal (i.e., $s = ((t_1, \ldots, t_k)_\theta; (\mathcal{G}, \mathcal{U}))$), and let $\mathcal{V}(s) = \mathcal{V}(t_1) \cup \ldots \cup \mathcal{V}(t_k)$. We define*

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

$$enc^{out}(s) = \begin{cases} enc^{out}(Succ_1(s))\,\mu, & \text{if } s \in Inst(Gr) \text{ where } \mu \text{ is associated to } s \\ f^{out}_s(\mathcal{G}^{out}(s)), & \text{otherwise, where } \mathcal{G}^{out}(s) = NextG((t_1, \ldots, t_k), \mathcal{G}) \setminus \mathcal{G} \end{cases}$$

Here, we extended $NextG$ to work also on queries:

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

So to compute $NextG((t_1, \ldots, t_k), \mathcal{G})$ for a query $(t_1, \ldots, t_k)$, in the beginning we only know that the variables in $\mathcal{G}$ represent ground terms. Then we compute the variables $NextG(t_1, \mathcal{G})$ which are made ground by all answer substitutions for concretizations of t_1. Next, we compute $NextG(t_2,\, NextG(t_1, \mathcal{G}))$ which are made ground by all answer substitutions for concretizations of t_2, etc.

Now we encode the paths of Gr as rewrite rules. However, we only consider *connection paths* of Gr, which suffice to analyze termination. Connection paths are non-empty paths that start in the root node of the graph or in a successor 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, G, and I. 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 A, E, F, H, I, J, K, but apart from E and J, connection paths may not traverse any of these end nodes in between.

Thus, we have connection paths from A to E, A to F, G to I, I to J, and I to K. These paths cover all ways through the graph except for INST edges (which are covered by the encoding of states to terms), for SPLIT edges (which we consider later in Def. 15), and for graph parts without cycles or SUC nodes (which cannot cause non-termination).

DEFINITION 13 (Connection Path). *A path $\pi = s_1 \ldots s_k$ is a* connection path *of an evaluation graph Gr iff $k > 1$ and*

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

For a connection path π, let σ_π represent the unifiers that were applied along the path. These unifiers can be determined by "comparing" the substitution labels of the first and the last state of the path (i.e., the goal in π's first state has a substitution label θ and the first goal of π's last state is labeled by $\theta\sigma_\pi$). So for the connection path π from A to F we have $\sigma_\pi = \sigma_5$, where $\sigma_5(\overline{T_1}) = \overline{T_5}$ and $\sigma_5(\overline{T_2}) = \overline{T_6}$. For this path, we generate rewrite rules which evaluate the instantiated input term $enc^{in}(\text{A})\,\sigma_\pi$ for the start node A to its output term $enc^{out}(\text{A})\,\sigma_\pi$ if the input term $enc^{in}(\text{F})$ for the end node can be evaluated to its output term $enc^{out}(\text{F})$. So we get $enc^{in}(\text{A})\,\sigma_\pi \to \mathsf{u}_{\text{A,F}}(\,enc^{in}(\text{F}), \mathcal{V}(enc^{in}(\text{A})\,\sigma_\pi)\,)$ and $\mathsf{u}_{\text{A,F}}(\,enc^{out}(\text{F}),\, \mathcal{V}(enc^{in}(\text{A})\,\sigma_\pi)\,) \to enc^{out}(\text{A})\,\sigma_\pi$ for a fresh function symbol $\mathsf{u}_{\text{A,F}}$. In our example, this yields

$$f^{in}_{\text{A}}(T_5, T_6) \to \mathsf{u}_{\text{A,F}}(f^{in}_{\text{F}}(T_5, T_6), T_5, T_6) \tag{7}$$

$$\mathsf{u}_{\text{A,F}}(f^{out}_{\text{F}}(T_7), T_5, T_6) \to f^{out}_{\text{A}} \tag{8}$$

However, for connection paths π' like the one from A to E which end in a SUC node, the resulting rewrite rule directly evaluates the instantiated input term $enc^{in}(\text{A})\,\sigma_{\pi'}$ for the start node A to its output term $enc^{out}(\text{A})\,\sigma_{\pi'}$. So we obtain

$$f^{in}_{\text{A}}(T_3, [\,]) \to f^{out}_{\text{A}} \tag{9}$$

DEFINITION 14 (Rules for Connection Paths). *Let π be a connection path $s_1 \ldots s_k$ in a symbolic evaluation graph. Let the (only) goal in s_1 be labeled by the substitution θ and let the first goal in s_k be labeled by the substitution $\theta\,\sigma_\pi$. If $s_k \in Suc(Gr)$, then we define $ConnectionRules(\pi) = \{enc^{in}(s_1)\sigma_\pi \to enc^{out}(s_1)\,\sigma_\pi\}$. Otherwise, $ConnectionRules(\pi)$ =*

$$\{\; enc^{in}(s_1)\,\sigma_\pi \;\to\; \mathsf{u}_{s_1,s_k}(\,enc^{in}(s_k),\, \mathcal{V}(enc^{in}(s_1)\,\sigma_\pi)\,),\;\; \mathsf{u}_{s_1,s_k}(\,enc^{out}(s_k),\, \mathcal{V}(enc^{in}(s_1)\,\sigma_\pi)\,) \;\to\; enc^{out}(s_1)\,\sigma_\pi \;\},$$

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

In addition to the rules for connection paths, we also need rewrite rules to simulate the evaluation of SPLIT nodes like F. Let δ be the substitution associated to F (i.e., δ represents the answer substitution of F's first successor G). Then the SPLIT node F succeeds (i.e., $enc^{in}(\text{F})\,\delta$ can be evaluated to $enc^{out}(\text{F})\,\delta$) if both successors G and H succeed (i.e., $enc^{in}(\text{G})\,\delta$ can be evaluated to $enc^{out}(\text{G})\,\delta$ and $enc^{in}(\text{H})$ can be evaluated to $enc^{out}(\text{H})$). Note that $enc^{in}(\text{F})$ and $enc^{in}(\text{G})$ only contain "input" arguments (i.e., abstract variables from $\mathcal{G}$) and thus, δ does not modify them. Hence, $enc^{in}(\text{F})\,\delta = enc^{in}(\text{F})$ and $enc^{in}(\text{G})\,\delta = enc^{in}(\text{G})$. So we obtain

$$f^{in}_{\text{F}}(T_5, T_6) \to \mathsf{u}_{\text{F,G}}(f^{in}_{\text{G}}(T_5, T_6), T_5, T_6) \tag{10}$$

$$\mathsf{u}_{\text{F,G}}(f^{out}_{\text{G}}(T_8), T_5, T_6) \to \mathsf{u}_{\text{G,H}}(f^{in}_{\text{A}}(T_5, T_8), T_5, T_6, T_8) \tag{11}$$

$$\mathsf{u}_{\text{G,H}}(f^{out}_{\text{A}}, T_5, T_6, T_8) \to f^{out}_{\text{F}}(T_8) \tag{12}$$

DEFINITION 15 (Rules for Split, $\mathcal{R}(Gr)$). *Let $s \in Split(Gr)$, $s_1 = Succ_1(s)$, and $s_2 = Succ_2(s)$. Moreover, let δ be the substitution associated to s. Then $SplitRules(s)$ =*

$$\begin{aligned} \{\; & enc^{in}(s) \;\to\; \mathsf{u}_{s,s_1}(\,enc^{in}(s_1),\, \mathcal{V}(enc^{in}(s))\,),\\ & \mathsf{u}_{s,s_1}(\,enc^{out}(s_1)\,\delta,\, \mathcal{V}(enc^{in}(s))\,) \;\to\; \mathsf{u}_{s_1,s_2}(\,enc^{in}(s_2),\, \mathcal{V}(enc^{in}(s)) \cup \mathcal{V}(enc^{out}(s_1)\delta)\,),\\ & \mathsf{u}_{s_1,s_2}(enc^{out}(s_2), \mathcal{V}(enc^{in}(s)) \cup \mathcal{V}(enc^{out}(s_1)\,\delta)) \to enc^{out}(s)\delta \;\} \end{aligned}$$

$\mathcal{R}(Gr)$ consists of $ConnectionRules(\pi)$ for all connection paths π and of $SplitRules(s)$ for all SPLIT *nodes s of Gr.*

For the graph Gr of Fig. 6, the resulting TRS $\mathcal{R}(Gr)$ consists of (7) – (12) and the connection rules (13), (14) for the path from G to I (where $\sigma_6(\overline{T_5}) = [\overline{T_9} \mid \overline{T_{10}}], \sigma_6(T_7) = T_{11}, \sigma_6(\overline{T_6}) = [\overline{T_9} \mid \overline{T_{12}}]$), the rules (15), (16) for I to K (where $\sigma_9(\overline{T_{10}}) = [\overline{T_{14}} \mid \overline{T_{15}}], \sigma_9(T_{11}) = T_{16}, \sigma_9(\overline{T_{12}}) = [\overline{T_{14}} \mid \overline{T_{17}}]$), and (17) for I to J (where $\sigma_8 = \sigma_7|_{\{\overline{T_{10}}, \overline{T_{12}}\}}$ with $\sigma_8(\overline{T_{10}}) = [\,], \sigma_8(\overline{T_{12}}) = \overline{T_{13}}$).

$$f_{\mathrm{G}}^{in}([T_9 \mid T_{10}], [T_9 \mid T_{12}]) \to \mathsf{u}_{\mathrm{G,I}}(f_{\mathrm{I}}^{in}(T_{10}, T_{12}), T_9, T_{10}, T_{12}) \quad (13)$$

$$\mathsf{u}_{\mathrm{G,I}}(f_{\mathrm{I}}^{out}(T_{11}), T_9, T_{10}, T_{12}) \to f_{\mathrm{G}}^{out}(T_{11}) \quad (14)$$

$$f_{\mathrm{I}}^{in}([T_{14} \mid T_{15}], [T_{14} \mid T_{17}]) \to \mathsf{u}_{\mathrm{I,K}}(f_{\mathrm{I}}^{in}(T_{15}, T_{17}), T_{14}, T_{15}, T_{17}) \quad (15)$$

$$\mathsf{u}_{\mathrm{I,K}}(f_{\mathrm{I}}^{out}(T_{16}), T_{14}, T_{15}, T_{17}) \to f_{\mathrm{I}}^{out}(T_{16}) \quad (16)$$

$$f_{\mathrm{I}}^{in}([\,], T_{13}) \to f_{\mathrm{I}}^{out}(T_{13}) \quad (17)$$

Thm. 16 states that the resulting TRS can simulate all successful evaluations represented in the graph, i.e., it simulates all computations of the logic program.

THEOREM 16 (TRS Simulates Semantics). *Let $s = (S; KB)$ be a start node of a connection path or a* SPLIT *node in a graph Gr, $S\gamma \in \mathcal{CON}(s)$, and let θ be an answer substitution for $S\gamma$. Then $enc^{in}(s)\gamma \xrightarrow{\mathrm{i}}^{+}_{\mathcal{R}(Gr)} enc^{out}(s)\gamma\theta$.*

Virtually all modern TRS termination tools can prove that $\mathcal{R}(Gr)$ is terminating in our example. Thm. 17 shows that this implies termination of all queries corresponding to the root of Gr. Hence, by our approach, one can prove termination of non-definite logic programs like Ex. 2 automatically.

THEOREM 17 (Soundness of Termination Analysis). *Let $\mathcal{P}$ be a logic program, $\mathsf{p} \in \Sigma$, m a moding function, and let Gr be a symbolic evaluation graph for $\mathcal{P}$ whose root is the initial state corresponding to $\mathcal{Q}_m^{\mathsf{p}}$. If the TRS $\mathcal{R}(Gr)$ is innermost terminating, then there is no infinite evaluation starting with any query from $\mathcal{Q}_m^{\mathsf{p}}$. Thus, all these queries are terminating w.r.t. the program $\mathcal{P}$.*

We implemented our approach for termination analysis in the tool AProVE [15]. In addition to the cut, our implementation handles many further features of Prolog. For our experiments, AProVE ran on all 477 Prolog programs of the *Termination Problem Database* (TPDB, version 8.0.6), which is the collection of examples used in the annual *International Termination Competition*.[5] 300 of them are definite logic programs, whereas the remaining 177 programs contain advanced features like cuts. 37 of the 477 examples are known to be non-terminating. The experiments were run on 2.2 GHz Quad-Opteron 848 Linux machines with a timeout of 60 seconds per program. In the table, "**Yes**" indicates the number of examples where termination could be proved and "**RT**" is the average runtime (in seconds) per example.

	Yes	**RT**
AProVE-[35]	265	7.1
AProVE-[37]	287	7.6
AProVE-[40]	340	5.7
AProVE-New	342	6.5

All termination tools for logic programs except AProVE ignore cuts, i.e., they try to prove termination of the program that results from removing the cuts. This is sensible, since cuts are not always needed for termination. Indeed, the variant AProVE-[35] implements our technique from [35] which ignores cuts and directly translates logic programs to TRSs. Still, it proves termination of 31 of the 177 non-definite programs. Other existing termination tools would not yield better results, as AProVE-[35] is already the most powerful tool for definite logic programs (as shown by the experiments in [35]) and as most of the remaining non-definite examples do not terminate anymore if one removes cuts. AProVE-[37] implements our approach from [37] which introduced evaluation graphs, but transforms them to definite logic programs instead of TRSs. This approach is much more powerful than [35] on examples with cut, but it fails on many definite logic programs where [35] was successful. The approach of the current paper (implemented in AProVE-New)[6] considers other paths in the graph than [37]. Thus, it simulates the evaluations of the original logic program more concisely and results in a more powerful approach (both for definite and non-definite programs).

[40] improved upon [37] by generating "dependency triples" from evaluation graphs. Indeed, AProVE-New and AProVE-[40] have almost the same power. But while the back-end of [40] required a tool that can handle the (non-standard) notion of dependency triples, our new approach works with any tool for termination of TRSs. Moreover, the approach of the current paper has the advantage that the TRSs generated for termination analysis can also be used for analyzing other properties like complexity, as shown in Sect. 5.

5. From Symbolic Evaluation Graphs to TRSs – Complexity Analysis

We briefly recapitulate the required notions for complexity of TRSs. 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". So for $\mathcal{R}(Gr)$ from Sect. 4, we have $\Sigma_d = \{f_{\mathrm{A}}^{in}, \mathsf{u}_{\mathrm{A,F}}, f_{\mathrm{F}}^{in}, \mathsf{u}_{\mathrm{F,G}}, \mathsf{u}_{\mathrm{G,H}}, f_{\mathrm{G}}^{in}, \mathsf{u}_{\mathrm{G,I}}, f_{\mathrm{I}}^{in}, \mathsf{u}_{\mathrm{I,K}}\}$. Different notions of complexity have been proposed for TRSs. In this paper, we focus on *innermost runtime complexity* [21], which corresponds to the notion of complexity used for programming languages. Here, one only considers rewrite sequences starting with *basic* terms $\mathsf{f}(t_1, \ldots, t_n)$, where $\mathsf{f} \in \Sigma_d$ and $t_1, \ldots, t_n$ do not contain symbols from Σ_d. The *innermost runtime complexity function* $irc_{\mathcal{R}}$ maps any $n \in \mathbb{N}$ to the length of the longest sequence of $\xrightarrow{\mathrm{i}}_{\mathcal{R}}$-steps starting with a basic term t where $|t| \le n$. 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 growth 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. Tools for automated complexity analysis of TRSs can automatically determine $irc_{\mathcal{R}(Gr)}(n) \in \mathcal{O}(n)$ for $\mathcal{R}(Gr) = \{(7) - (17)\}$ from Sect. 4.[7]

Moreover, we also have to define the notion of "complexity" for logic programs. For a logic program $\mathcal{P}$ and a query Q, we consider the length of the longest evaluation starting in the initial state for Q. As shown in [41], this length is equal to the number of unification attempts when traversing the whole SLD tree according to the ISO semantics [23], up to a constant factor.[8] For a moding function m, and any term $\mathsf{p}(t_1, \ldots, t_n)$, its *moded size* is $|\mathsf{p}(t_1, \ldots, t_n)|_m = 1 + \Sigma_{i \in \{i \mid 1 \le i \le n, m(\mathsf{p}, i) = in\}} |t_i|$. Thus, for 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 evaluation starting with the initial state for some query $Q \in \mathcal{Q}_m^{\mathsf{p}}$ with $|Q|_m \le n$.

To analyze $prc_{\mathcal{P}, \mathcal{Q}_m^{\mathsf{p}}}(n)$, we generate an evaluation graph Gr for $\mathcal{Q}_m^{\mathsf{p}}$ as in Sect. 3 and obtain the TRS $\mathcal{R}(Gr)$ as in Sect. 4. At first sight, one might expect that asymptotically, $irc_{\mathcal{R}(Gr)}(n)$ is indeed an upper bound of $prc_{\mathcal{P}, \mathcal{Q}_m^{\mathsf{p}}}(n)$. This would allow us to use

[5] In these competitions, AProVE was the most powerful tool for termination of logic programs, see `http://termination-portal.org/wiki/Termination_Competition/`.

[6] To benefit from the full power of rewriting-based termination analysis, in our implementation we generate TRSs together with an *argument filtering*, as in [35]. In this way, one can also handle examples where ground information on the arguments of predicates is not sufficient.

[7] For example, this can be determined by the tool TCT [3]. While AProVE was the most powerful tool for innermost runtime complexity analysis in the recent termination competitions, here it only obtains $irc_{\mathcal{R}(Gr)}(n) \in \mathcal{O}(n^2)$.

[8] In contrast, other approaches like [10–12, 30] use the number of resolution steps to measure complexity. As long as one does not consider dynamic built-in predicates like assert/1, these measures are asymptotically equivalent, as the number of unification attempts at each resolution step is bounded by a constant (i.e., by the number of program clauses).

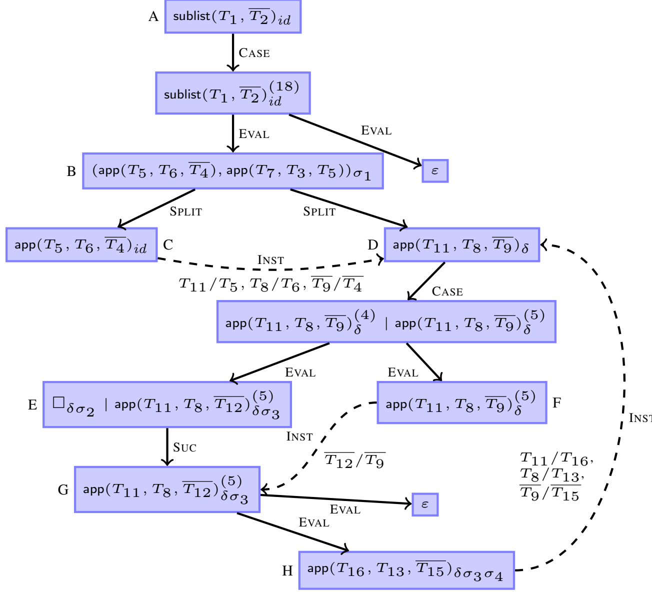


Figure 19. Symbolic Evaluation Graph for Ex. 18

existing methods for complexity analysis of TRSs in order to derive upper bounds on the runtime of logic programs.

In fact for Ex. 2, both $irc_{\mathcal{R}(Gr)}(n)$ and $prc_{\mathcal{P},\mathcal{Q}^{\mathsf{star}}_m}(n)$ are in $\mathcal{O}(n)$, i.e., the complexity of the logic program for $\mathcal{Q}^{\mathsf{star}}_m$ is also linear. But in general, $irc_{\mathcal{R}(Gr)}(n)$ is not necessarily an upper bound of $prc_{\mathcal{P},\mathcal{Q}^{\mathsf{p}}_m}(n)$. This can happen if Gr contains a SPLIT node whose first successor is *not deterministic*. A query Q is *deterministic* iff it generates at most one answer substitution at most once [25]. Similarly, we call an abstract state s *deterministic* iff each of its concretizations has at most one evaluation to a state of the form $(\Box_\theta \mid S)$.

EXAMPLE 18. *To see the problems with* SPLIT *nodes whose first successor is not deterministic, consider the following program from the TPDB which consists of the clauses (4) and (5) for* app *and the following rule:*

$$\mathsf{sublist}(X, Y) \text{ :- } \mathsf{app}(P, U, Y),\ \mathsf{app}(V, X, P). \tag{18}$$

We regard the class of queries $\mathcal{Q}^{\mathsf{sublist}}_m$, where $m(\mathsf{sublist}, 1) = out$ and $m(\mathsf{sublist}, 2) = in$. The program computes (by backtracking) all sublists of a given list. Its complexity is quadratic since the first app*-call results in a linear evaluation with a linear number of solutions. The second* app*-call again needs linear time, but due to backtracking, it is called linearly often.*

We obtain the evaluation graph Gr in Fig. 19. For readability, we omitted labels $t \nsim t'$ on EVAL*-edges. We have $\sigma_1(T_1) = T_3, \sigma_1(\overline{T_2}) = \overline{T_4}$; $\sigma_2(T_8) = \overline{T_{12}}, \sigma_2(\overline{T_9}) = \overline{T_{12}}, \sigma_2(T_{11}) = [\,]$; $\sigma_3(\overline{T_9}) = \overline{T_{12}}$; $\sigma_4(T_8) = T_{13}, \sigma_4(\overline{T_{12}}) = [\overline{T_{14}} \mid \overline{T_{15}}], \sigma_4(T_{11}) = [\overline{T_{14}} \mid T_{16}]$; and $\delta(T_3) = T_8, \delta(T_5) = \overline{T_9}, \delta(T_6) = \overline{T_{10}}, \delta(T_7) = T_{11}$.*

This symbolic evaluation graph has connection paths from A *to* B, D *to* E, D *to* G, D *to* F, *and* G *to* H. *It gives rise to the following TRS $\mathcal{R}(Gr)$.*

$$f^{in}_{\mathrm{A}}(T_4) \to \mathsf{u}_{\mathrm{A,B}}(f^{in}_{\mathrm{B}}(T_4), T_4) \tag{19}$$

$$\mathsf{u}_{\mathrm{A,B}}(f^{out}_{\mathrm{B}}(T_5, T_6, T_7, T_3), T_4) \to f^{out}_{\mathrm{A}}(T_3) \tag{20}$$

$$f^{in}_{\mathrm{B}}(T_4) \to \mathsf{u}_{\mathrm{B,C}}(f^{in}_{\mathrm{D}}(T_4), T_4) \tag{21}$$

$$\mathsf{u}_{\mathrm{B,C}}(f^{out}_{\mathrm{D}}(T_9, T_{10}), T_4) \to \mathsf{u}_{\mathrm{C,D}}(f^{in}_{\mathrm{D}}(T_9), T_4, T_9, T_{10}) \tag{22}$$

$$\mathsf{u}_{\mathrm{C,D}}(f^{out}_{\mathrm{D}}(T_{11}, T_8), T_4, T_9, T_{10}) \to f^{out}_{\mathrm{B}}(T_9, T_{10}, T_{11}, T_8) \tag{23}$$

$$f^{in}_{\mathrm{D}}(T_{12}) \to f^{out}_{\mathrm{D}}([\,], T_{12}) \tag{24}$$

$$f^{in}_{\mathrm{D}}(T_{12}) \to \mathsf{u}_{\mathrm{D,G}}(f^{in}_{\mathrm{G}}(T_{12}), T_{12}) \tag{25}$$

$$\mathsf{u}_{\mathrm{D,G}}(f^{out}_{\mathrm{G}}(T_{11}, T_8), T_{12}) \to f^{out}_{\mathrm{D}}(T_{11}, T_8) \tag{26}$$

$$f^{in}_{\mathrm{D}}(T_9) \to \mathsf{u}_{\mathrm{D,F}}(f^{in}_{\mathrm{G}}(T_9), T_9) \tag{27}$$

$$\mathsf{u}_{\mathrm{D,F}}(f^{out}_{\mathrm{G}}(T_{11}, T_8), T_9) \to f^{out}_{\mathrm{D}}(T_{11}, T_8) \tag{28}$$

$$f^{in}_{\mathrm{G}}([T_{14} \mid T_{15}]) \to \mathsf{u}_{\mathrm{G,H}}(f^{in}_{\mathrm{D}}(T_{15}), T_{14}, T_{15}) \tag{29}$$

$$\mathsf{u}_{\mathrm{G,H}}(f^{out}_{\mathrm{D}}(T_{16}, T_{13}), T_{14}, T_{15}) \to f^{out}_{\mathrm{G}}([T_{14} \mid T_{16}], T_{13}) \tag{30}$$

Its termination is easy to prove by tools like AProVE*, which implies termination of the logic program by Thm. 17. However, this TRS cannot be used for complexity analysis, as $irc_{\mathcal{R}(Gr)}$ is linear whereas the runtime complexity of the original logic program is quadratic. For an analogous reason, complexity analysis of such examples is also not possible by transformations from logic programs to TRSs like [33, 35].*

For complexity analysis, we need a more sophisticated treatment of SPLIT *nodes than for termination analysis. 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. However for complexity analysis, 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. 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 (i.e., of* D*) is repeated k times. This is not simulated by the TRS, which replaces backtracking by non-deterministic choice. So after applying rule (21), one has to perform a "first f^{in}_{D}-reduction" to evaluate the f^{in}_{D}-term in the right-hand side to a f^{out}_{D}-term. There exist several possibilities for this reduction (e.g., by using (24), (25), or (27)). So one chooses one such reduction non-deterministically. Afterwards, the remaining rewrite sequence continues with rule (22). However, the TRS does not reflect that in the logic program, one would backtrack afterwards and repeat this remaining rewrite sequence with rule (22), for* every *possible "first f^{in}_{D}-reduction" from $f^{in}_{\mathrm{D}}(\dots)$ to $f^{out}_{\mathrm{D}}(\dots)$.*

However, for the star-example of Ex. 2, the first successor G of the only SPLIT node F in the graph of Fig. 6 is deterministic. The reason is that there is at most one answer substitution for any query $\mathsf{app}(t_5, t_7, t_6)$, where t_5 and t_6 are ground terms. In Sect. 6, we will show how to use evaluation graphs in order to analyze determinacy automatically.

Nevertheless, even if all first successors of SPLIT nodes are deterministic, $irc_{\mathcal{R}(Gr)}$ is not necessarily an upper bound of $prc_{\mathcal{P},\mathcal{Q}^{\mathsf{p}}_m}$. This can happen if (i) a SPLIT node s can reach itself via a non-empty path, (ii) its first successor s' reaches a SUC node s'', and (iii) s'' reaches a cycle in the graph.

EXAMPLE 21. *Consider the following program $\mathcal{P}$ and the set of queries $\mathcal{Q}^{\mathsf{a}}_m$ where $m(\mathsf{a}, 1) = in$.*

$$\begin{aligned} \mathsf{a}(X) &\text{ :- } \mathsf{b}(X), \mathsf{q}(X).\\ \mathsf{b}(X)&.\\ \mathsf{b}(X) &\text{ :- } \mathsf{p}(X).\\ \mathsf{p}(\mathsf{s}(X)) &\text{ :- } \mathsf{p}(X).\\ \mathsf{q}(\mathsf{s}(X)) &\text{ :- } \mathsf{a}(X). \end{aligned}$$

In the corresponding symbolic evaluation graph in Fig. 20, dotted arrows abbreviate paths of several edges. We have $\sigma_1(\overline{T_1}) = \overline{T_2}$, $\sigma_2(\overline{T_2}) = \overline{T_3}$, $\sigma_3(\overline{T_3}) = \overline{T_4}$, $\sigma_4(\overline{T_4}) = \mathsf{s}(\overline{T_5})$, and $\sigma_5(\overline{T_2}) =$

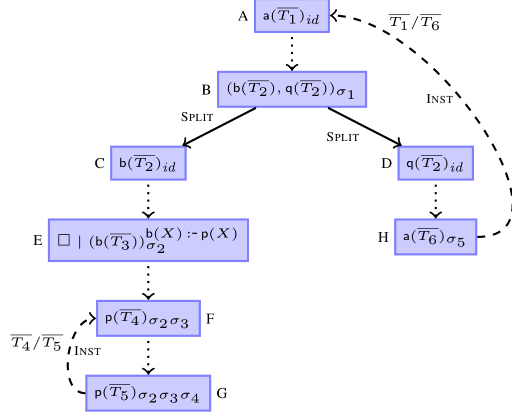


Figure 20. Symbolic Evaluation Graph for Ex. 21

$\mathsf{s}(\overline{T_6})$. *Here, (i) the* SPLIT *node* B *reaches itself via a non-empty path, (ii) its first successor* C *reaches a* SUC *node* E*, and (iii)* E *reaches another cycle (from* F *to* G*). The graph has connection paths from* A *to* B*,* C *to* E*,* C *to* F*,* F *to* G*, and* D *to* H*. It results in the following TRS.*

$$f^{in}_{\mathrm{A}}(T_2) \;\to\; \mathsf{u}_{\mathrm{A,B}}(f^{in}_{\mathrm{B}}(T_2), T_2) \quad (31)$$
$$\mathsf{u}_{\mathrm{A,B}}(f^{out}_{\mathrm{B}}, T_2) \;\to\; f^{out}_{\mathrm{A}} \quad (32)$$
$$f^{in}_{\mathrm{B}}(T_2) \;\to\; \mathsf{u}_{\mathrm{B,C}}(f^{in}_{\mathrm{C}}(T_2), T_2) \quad (33)$$
$$\mathsf{u}_{\mathrm{B,C}}(f^{out}_{\mathrm{C}}, T_2) \;\to\; \mathsf{u}_{\mathrm{C,D}}(f^{in}_{\mathrm{D}}(T_2), T_2) \quad (34)$$
$$\mathsf{u}_{\mathrm{C,D}}(f^{out}_{\mathrm{D}}, T_2) \;\to\; f^{out}_{\mathrm{B}} \quad (35)$$
$$f^{in}_{\mathrm{C}}(T_3) \;\to\; f^{out}_{\mathrm{C}} \quad (36)$$
$$f^{in}_{\mathrm{C}}(T_4) \;\to\; \mathsf{u}_{\mathrm{C,F}}(f^{in}_{\mathrm{F}}(T_4), T_4) \quad (37)$$
$$\mathsf{u}_{\mathrm{C,F}}(f^{out}_{\mathrm{F}}, T_4) \;\to\; f^{out}_{\mathrm{C}} \quad (38)$$
$$f^{in}_{\mathrm{F}}(\mathsf{s}(T_5)) \;\to\; \mathsf{u}_{\mathrm{F,G}}(f^{in}_{\mathrm{F}}(T_5), T_5) \quad (39)$$
$$\mathsf{u}_{\mathrm{F,G}}(f^{out}_{\mathrm{F}}, T_5) \;\to\; f^{out}_{\mathrm{F}} \quad (40)$$
$$f^{in}_{\mathrm{D}}(\mathsf{s}(T_6)) \;\to\; \mathsf{u}_{\mathrm{D,H}}(f^{in}_{\mathrm{A}}(T_6), T_6) \quad (41)$$
$$\mathsf{u}_{\mathrm{D,H}}(f^{out}_{\mathrm{A}}, T_6) \;\to\; f^{out}_{\mathrm{D}} \quad (42)$$

For the complexity $prc_{\mathcal{P},\mathcal{Q}^{\mathsf{a}}_m}$ of this program, each call to b *yields both a success (from* C *to* E *in constant time) and a failing further computation (by the cycle from* F *to* G *which takes linear time). Since* b *is called linearly often (by the cycle from* A *to* H*), we obtain a quadratic runtime in total.*

However, the resulting TRS only has linear complexity. Here, the backtracking after the SUC *node* E *is modeled by non-deterministic choice. So to evaluate an f^{in}_{C}-term, one* either *uses rule (36) which corresponds to the path from* C *to* E or *the rules (37), (38) which correspond to the path from* C *to* F, but not both. *The traversal of the cycle from* A *to* H *can only continue if one evaluates f^{in}_{C} by rule (36), which works in constant time. Only then can the right-hand side of (33) evaluate to the left-hand side of (34).*

Def. 22 captures when $irc_{\mathcal{R}(Gr)}$ is no upper bound of $prc_{\mathcal{P},\mathcal{Q}^{\mathsf{p}}_m}$.

DEFINITION 22 (Multiplicative SPLIT Nodes). *A* SPLIT *node s in a symbolic evaluation graph Gr is called* multiplicative *iff its first successor is not deterministic or if s satisfies the three conditions* (i) – (iii) *above. Let $mults(Gr)$ be the set of all multiplicative* SPLIT *nodes of Gr.*

The only SPLIT node F in the graph of Fig. 6 is indeed non-multiplicative. Its first successor G is deterministic and while F can reach itself via a non-empty path, the only SUC node reachable from its first successor G is J, but J cannot reach a cycle in Gr (i.e., (iii) does not hold).

Thm. 23 shows that if the symbolic evaluation graph only contains non-multiplicative SPLIT nodes, our approach can also be used for complexity analysis of logic programs. So the linear complexity of $\mathcal{R}(Gr)$ in our example indeed implies linear complexity of the original program from Ex. 2.

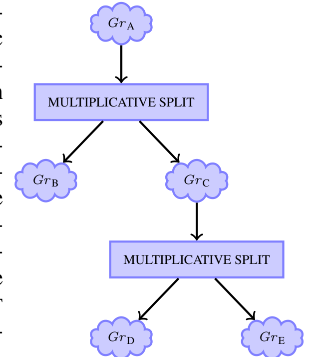
THEOREM 23 (Soundness of Complexity Analysis I). *Let $\mathcal{P}$ be a logic program, $\mathsf{p} \in \Sigma$, m a moding function, and let Gr be a symbolic evaluation graph for $\mathcal{P}$ whose root is the initial state corresponding to $\mathcal{Q}^{\mathsf{p}}_m$. If Gr has no multiplicative* SPLIT *nodes, then $prc_{\mathcal{P},\mathcal{Q}^{\mathsf{p}}_m}(n) \in \mathcal{O}(irc_{\mathcal{R}(Gr)}(n))$.*

We now extend our approach to also handle examples like Ex. 18 where the evaluation graph Gr contains multiplicative SPLIT nodes (i.e., here we have $mults(Gr) = \{\mathrm{B}\}$).

To this end, we generate two *separate* TRSs $\mathcal{R}(Gr_{\mathrm{C}})$ and $\mathcal{R}(Gr_{\mathrm{D}})$ for the subgraphs starting in the two successors C and D of a multiplicative SPLIT node like B in Ex. 18, and multiply their complexity functions $irc_{\mathcal{R}(Gr_{\mathrm{C}}),\mathcal{R}(Gr)}$ and $irc_{\mathcal{R}(Gr_{\mathrm{D}}),\mathcal{R}(Gr)}$. Here, $irc_{\mathcal{R}(Gr_{\mathrm{C}}),\mathcal{R}(Gr)}$ differs from the ordinary complexity function $irc_{\mathcal{R}(Gr)}$ by only counting those rewrite steps that are done with the sub-TRS $\mathcal{R}(Gr_{\mathrm{C}}) \subseteq \mathcal{R}(Gr)$.

In general, for any $\mathcal{R}' \subseteq \mathcal{R}$, the function $irc_{\mathcal{R}',\mathcal{R}}$ maps any $n \in \mathbb{N}$ to the maximal number of $\xrightarrow{\mathrm{i}}_{\mathcal{R}'}$-steps that occur in any sequence of $\xrightarrow{\mathrm{i}}_{\mathcal{R}}$-steps starting with a basic term t where $|t| \leq n$. Related notions of "relative" complexity for TRSs were used in, e.g., [4, 21, 32, 42]. Most existing automated complexity provers can also approximate $irc_{\mathcal{R}',\mathcal{R}}$ asymptotically.

The function $irc_{\mathcal{R}(Gr_{\mathrm{C}}),\mathcal{R}(Gr)}$ indeed also yields an upper bound on the number 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. Hence, by multiplying the linear runtime complexities of $irc_{\mathcal{R}(Gr_{\mathrm{C}}),\mathcal{R}(Gr)}$ and $irc_{\mathcal{R}(Gr_{\mathrm{D}}),\mathcal{R}(Gr)}$, we obtain the correct result that the runtime of the original logic program is (at most) quadratic.



So we use the multiplicative SPLIT nodes of a symbolic evaluation graph Gr to decompose Gr into subgraphs, such that multiplicative SPLIT nodes only occur as the leaves of subgraphs. As an example, the symbolic evaluation graph on the side is decomposed into the subgraphs $Gr_{\mathrm{A}}, \dots, Gr_{\mathrm{E}}$ (the subgraphs Gr_{A} and Gr_{C} include the respective multiplicative SPLIT node as a leaf). We now determine the runtime complexities $irc_{\mathcal{R}(Gr_{\mathrm{A}}),\mathcal{R}(Gr)}, \dots, irc_{\mathcal{R}(Gr_{\mathrm{E}}),\mathcal{R}(Gr)}$ separately and combine them to obtain an upper bound for the runtime of the whole logic program. As discussed above, the runtime complexity functions resulting from subgraphs of a multiplicative SPLIT node have to be multiplied. In contrast, the runtimes for subgraphs above a multiplicative SPLIT node have to be added. So for the graph on the side, we obtain $irc_{\mathcal{R}(Gr_{\mathrm{A}}),\mathcal{R}(Gr)}(n) + irc_{\mathcal{R}(Gr_{\mathrm{B}}),\mathcal{R}(Gr)}(n) \cdot (irc_{\mathcal{R}(Gr_{\mathrm{C}}),\mathcal{R}(Gr)}(n) + irc_{\mathcal{R}(Gr_{\mathrm{D}}),\mathcal{R}(Gr)}(n) \cdot irc_{\mathcal{R}(Gr_{\mathrm{E}}),\mathcal{R}(Gr)}(n))$ as an approximation for the complexity of the logic program.

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

DEFINITION 24 (Decomposable Graphs). *A symbolic evaluation graph Gr is called* decomposable *iff there is no non-empty path from a node $s \in mults(Gr)$ to itself.*

The graph in Ex. 18 is decomposable. However, decomposability is a restriction and there are programs in the TPDB whose complexity we cannot analyze, because our graph construction yields a non-decomposable evaluation graph.[9] For instance, the graph in Ex. 21 is not decomposable.

For any node s, the *subgraph at node s* starts in s and stops when reaching multiplicative SPLIT nodes.

DEFINITION 25 (Subgraphs). *Let Gr be a decomposable evaluation graph with nodes V and edges E (i.e., $Gr = (V, E)$) and let $s \in V$. We define the* subgraph of Gr at node s *as the minimal graph $Gr_s = (V_s, E_s)$ with $s \in V_s$ that satisfies the following property: whenever $s_1 \in V_s \setminus mults(Gr)$ 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 symbolic evaluation graph into the subgraph at the root node and into the subgraphs at all successors of multiplicative SPLIT nodes. So the graph in Ex. 18 is decomposed into Gr_{A}, Gr_{C}, and Gr_{D}, where Gr_{A} contains the 4 nodes from A to B and to ε, Gr_{C} contains all other nodes, and Gr_{D} contains all nodes of Gr_{C} except C.

$\mathcal{R}(Gr_{\text{A}}) = \{(19) - (23)\}$ consists of $ConnectionRules(\pi)$ for the connection path π from A to B and of $SplitRules(\text{B})$. For both Gr_{C} and Gr_{D}, we get the same TRS, because C is an instance of D, i.e., $\mathcal{R}(Gr_{\text{C}}) = \mathcal{R}(Gr_{\text{D}}) = \{(24) - (30)\}$.

For the complexity of the original logic program, we 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 represented by the subgraph of Gr at node s.

DEFINITION 26 (Complexity for Subgraphs). *Let $Gr = (V, E)$ be decomposable. For any $s \in V$ and $n \in \mathbb{N}$, let*

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

So in Ex. 18, we obtain $cplx_{\text{A}}(n) =$

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

Thm. 27 states that combining the complexities of the TRSs as in Def. 26 indeed yields an upper bound for the complexity of the original logic program.

THEOREM 27 (Soundness of Complexity Analysis II). *Let $\mathcal{P}$ be a logic program, $\mathsf{p} \in \Sigma$, m a moding function, and let Gr be a symbolic evaluation graph for $\mathcal{P}$ whose root is the initial state corresponding to Q_m^{p}. If Gr is decomposable, then we have $prc_{\mathcal{P},\mathcal{Q}_m^{\mathsf{p}}}(n) \in \mathcal{O}(cplx_{root(Gr)}(n))$.*

For Ex. 18, tools for complexity analysis of TRSs like TCT and AProVE automatically prove $irc_{\mathcal{R}(Gr_{\text{A}}),\mathcal{R}(Gr)}(n) \in \mathcal{O}(n)$,[10] $irc_{\mathcal{R}(Gr_{\text{C}}),\mathcal{R}(Gr)}(n) \in \mathcal{O}(n)$, $irc_{\mathcal{R}(Gr_{\text{D}}),\mathcal{R}(Gr)}(n) \in \mathcal{O}(n)$. This implies $cplx_{\text{A}}(n) = irc_{\mathcal{R}(Gr_{\text{A}}),\mathcal{R}(Gr)}(n) + irc_{\mathcal{R}(Gr_{\text{C}}),\mathcal{R}(Gr)}(n) \cdot irc_{\mathcal{R}(Gr_{\text{D}}),\mathcal{R}(Gr)}(n) \in \mathcal{O}(n^2)$. Thus, also $prc_{\mathcal{P},\mathcal{Q}_m^{\mathsf{sublist}}}(n) \in \mathcal{O}(n^2)$.

Note that Thm. 27 subsumes Thm. 23. Every evaluation graph Gr without multiplicative SPLIT nodes is decomposable and here we have $cplx_{root(Gr)}(n) = irc_{\mathcal{R}(Gr)}(n)$.

We also implemented our approach for complexity analysis in our tool AProVE [15]. Existing approaches for direct complexity analysis of logic programs (e.g., [10–12, 24, 30])[11] are restricted to well-moded logic programs. In contrast, our approach is applicable to a much wider class of logic programs (including non-well-moded and non-definite programs).[12] To compare their power, we evaluated AProVE against the Complexity Analysis System for LOGic (CASLOG) [11] and the Ciao Preprocessor (CiaoPP) [19, 20], which implements the approach of [30]. We ran the three tools on all 477 Prolog programs from the TPDB, again using 2.2 GHz Quad-Opteron 848 Linux machines with a timeout of 60 seconds per program. For CiaoPP we used both the original cost analysis (CiaoPP-o) and CiaoPP's new resource framework which allows to measure different forms of costs (CiaoPP-r). 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.

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \cdot 2^n)$	**bounds**	**RT**
CASLOG	1	21	4	**3**	29	14.8
CiaoPP-o	3	19	4	**3**	29	11.7
CiaoPP-r	3	18	4	**3**	28	12.5
AProVE	**54**	**117**	**37**	0	**208**	**10.6**

In the above table, we used one row for each tool. The first four columns 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 ($\mathcal{O}(n \cdot 2^n)$).[13] In column 5 and 6 we give the total number of upper bounds that could be found by the tool and its average runtime on each example. We highlight the best tool for each column using bold font.

The table shows that AProVE can find upper bounds for a much larger subset (42.8%) of the programs than any of the other tools (6.1%). Nevertheless, there are 6 examples where CASLOG or CiaoPP can prove constant (1), linear (1), quadratic (1), or exponential bounds (3), whereas AProVE fails. 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.

---

[9] An extension of our method to examples with non-decomposable evaluation graphs would be an interesting topic for further work. However, even with the restriction to decomposable graphs, our approach is substantially more powerful than all previous techniques for automated complexity analysis of logic programs, cf. the end of this section. In our experiments, there were only 3 examples where other tools could prove an (exponential) upper bound while we failed because of non-decomposability.

[10] We even have $irc_{\mathcal{R}(Gr_{\text{A}}),\mathcal{R}(Gr)}(n) \in \mathcal{O}(1)$, i.e., as in Footnote 7, the bounds found by the tools are not always tight.

[11] Some approaches also deduce *lower* complexity bounds for logic programs [12, 24], while we only infer *upper* bounds.

[12] However, our implementation currently does not treat built-in integer arithmetic, while [10–12, 30] handle linear arithmetic constraints. But our approach could be extended by generating TRSs with built-in integers [14] from the evaluation graphs. This was also done in our approaches for termination analysis of Java via term rewriting [7, 9].

[13] The back-end of AProVE for complexity analysis of TRSs currently only implements techniques for detecting polynomial bounds. When extending the TRS back-end by other techniques like [5], we could also infer exponential bounds.

6. Symbolic Evaluation Graphs for Determinacy Analysis

Finally, after having shown how symbolic evaluation graphs can be used for termination and complexity analysis, we consider a third kind of analysis, viz. *determinacy analysis* (cf. the definition of "determinacy" before Ex. 18). Several approaches for determinacy analysis have been developed (e.g., [25–29, 34]). Moreover, determinacy analysis is also needed for complexity analysis to detect non-deterministic SPLIT nodes in Thm. 23 and 27.

Every successful evaluation corresponds to a path to a SUC node in the evaluation graph. Therefore, this graph is very well suited as a basis for determinacy analysis. A sufficient criterion for determinacy of a state s in the graph is if there is no path starting in s which traverses more than one SUC node. In other words, if s reaches a SUC node s', then there may be no further non-empty path from s' to a SUC node.

THEOREM 28 (Soundness of Determinacy Criterion). *Let $\mathcal{P}$ be a logic program and let Gr be a symbolic evaluation graph for $\mathcal{P}$. Let s be a node in Gr such that for all* SUC *nodes s' reachable from s, there is no non-empty path from s' to a* SUC *node. Then s is deterministic. Thus, if s is the initial state corresponding to $\mathcal{Q}_m^{\mathsf{p}}$ for a $\mathsf{p} \in \Sigma$ and a moding function m, then all queries in $\mathcal{Q}_m^{\mathsf{p}}$ are also deterministic.*

For example, all nodes in the evaluation graph of Fig. 6 satisfy the above determinacy criterion, since there are no non-empty paths from the two SUC nodes E or J to a SUC node again. So the first successor G of the SPLIT node F is deterministic and thus, F is not multiplicative.

In contrast, the node C of the graph in Ex. 18 does not satisfy the determinacy criterion, since it reaches E which has a non-empty cycle to itself. Indeed, C is not deterministic and the corresponding SPLIT node B is multiplicative.

Finally, the nodes in the evaluation graph of Ex. 21 are again deterministic, since the only SUC node E has no non-empty path to itself. But since the SPLIT node B satisfies the conditions (i) – (iii), it is nevertheless multiplicative.

Our experiments in Sect. 5 indicate that the criterion of Thm. 28 is strong enough to detect non-multiplicative SPLIT nodes for complexity analysis. But in general, this criterion only represents a first step towards determinacy analysis based on symbolic evaluation graphs and several additional sufficient criteria for determinacy would be possible.

This is also indicated by our experiments when comparing the implementation of our determinacy analysis in AProVE with the determinacy analysis implemented in CiaoPP [28].[14] We again tested both tools on all 477 logic programs from the TPDB. On definite programs, CiaoPP was clearly more powerful (it proved determinacy for 132 out of 300 programs, whereas AProVE only succeeded for 19 programs). But on non-definite programs, AProVE's determinacy analysis is stronger (here, AProVE showed determinacy of 75 out of 177 examples, whereas CiaoPP only succeeded for 61 programs). Altogether, our new determinacy criterion based on evaluation graphs is a substantial addition to existing determinacy analyses, since AProVE succeeded on 58 examples where CiaoPP failed. In other words, by coupling our new technique with existing ones, the power of determinacy analysis can be increased significantly.

[14] We did not compare with the determinacy analyzer spdet implemented in SICStus Prolog 4.2.1, since it reports both false positives and false negatives.

7. Conclusion

We presented the symbolic evaluation graph and the use of term rewriting as a general methodology for the analysis of logic programs. These graphs represent all evaluations of a (possibly non-definite) logic program in a finite way. Therefore, they can be used as the basis for many different kinds of analyses. In particular, one can translate their paths to rewrite rules and use existing techniques from term rewriting to analyze the termination and complexity of the original logic program. Moreover, one can also perform analyses directly on the evaluation graph (e.g., to examine determinacy).

The current paper does not only give an overview on our previous work on this topic, but it introduces numerous new results. In Sect. 3, we presented a new formulation of the abstract inference rules which is suitable for the subsequent generation of TRSs. Moreover, the theorems of this section (on the connection between concrete and abstract evaluation rules) are new contributions. The approach for termination analysis in Sect. 4 is also substantially different from our earlier approaches, because it directly generates TRSs from evaluation graphs. In particular, this allows us to use the same approach for both termination and complexity analysis. The contributions in Sect. 5 and Sect. 6 (on complexity and determinacy analysis) are completely new.

We implemented all our results in the tool AProVE. Our experiments show that our approaches to termination and complexity analysis are more powerful than previous ones and that our approach to determinacy analysis is a substantial addition to existing ones. See [1] for further details on the experiments and to run AProVE via a web interface.[15]

Acknowledgments

We thank M. Hermenegildo and P. López-García for their support. Without it, the experimental comparisons 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 [1].

References

[1] http://aprove.informatik.rwth-aachen.de/eval/LPGraphs/.

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

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

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

[5] M. Avanzini, N. Eguchi, and G. Moser. A path order for rewrite systems that compute exponential time functions. In *Proc. RTA '11*, LIPIcs 10, pages 123–138, 2011.

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

[7] M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *Proc. RTA '11*, LIPIcs 10, pages 155–170, 2011.

[8] M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated detection of non-termination and NullPointerExceptions for Java Bytecode. In *Proc. FoVeOOS '11*, LNCS 7421, pages 123–141, 2012.

[9] M. Brockschmidt, R. Musiol, C. Otto, and J. Giesl. Automated termination proofs for Java programs with cyclic data. In *Proc. CAV '12*, LNCS 7358, pages 105–122, 2012.

[10] S. K. Debray, N.-W. Lin, and M. V. Hermenegildo. Task granularity analysis in logic programs. In *Proc. PLDI '90*, pages 174–188. ACM Press, 1990.

[15] [1] also contains a version of the paper with all proofs [18].

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

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

[13] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1–2), pages 69–116, 1987.

[14] C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *Proc. RTA '09*, LNCS 5595, pages 32–47, 2009.

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

[16] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3), pages 155–203, 2006.

[17] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Transactions on Programming Languages and Systems*, 33(2), 2011.

[18] J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic evaluation graphs and term rewriting – a general methodology for analyzing logic programs. Technical Report AIB 2012-12, RWTH Aachen University, 2012. Available from `http://aib.informatik.rwth-aachen.de` and [1].

[19] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Science of Computer Programming*, 58(1-2):115–140, 2005.

[20] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12:219–252, 2012.

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

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

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

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

[25] A. King, L. Lu, and S. Genaim. Detecting determinacy in Prolog programs. In *Proc. ICLP '06*, LNCS 4079, pages 132–147, 2006.

[26] J. Kriener and A. King. RedAlert: Determinacy inference for Prolog. In *Proc. ICLP '11, Theory and Practice of Logic Programming*, 11(4-5):537–553, 2011.

[27] J. Kriener and A. King. Mutual exclusion by interpolation. In *Proc. FLOPS '12*, LNCS 7294, pages 182–196, 2012.

[28] P. López-García, F. Bueno, and M. V. Hermenegildo. Automatic inference of determinacy and mutual exclusion for logic programs using mode and type analyses. *New Generation Computing*, 28(2):177–206, 2010.

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

[30] J. A. Navas, E. Mera, P. López-García, and M. V. Hermenegildo. User-definable resource bounds analysis for logic programs. In *Proc. ICLP '07*, LNCS 4670, pages 348–363, 2007.

[31] M. T. Nguyen, J. Giesl, and P. Schneider-Kamp. Termination analysis of logic programs based on dependency graphs. In *Proc. LOPSTR '07*, LNCS 4915, pages 8–22, 2008.

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

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

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

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

[36] P. Schneider-Kamp, J. Giesl, and M. T. Nguyen. The dependency triple framework for termination of logic programs. In *Proc. LOPSTR '09*, LNCS 6037, pages 37–51, 2010.

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

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

[39] T. Ströder. Towards termination analysis of real Prolog programs. Diploma Thesis, RWTH Aachen University, 2010. Available from [1].

[40] T. Ströder, P. Schneider-Kamp, and J. Giesl. Dependency triples for improving termination analysis of logic programs with cut. In *Proc. LOPSTR '10*, LNCS 6564, pages 184–199, 2011.

[41] T. Ströder, F. Emmes, P. Schneider-Kamp, J. Giesl, and C. Fuhs. A linear operational semantics for termination and complexity analysis of ISO Prolog. In *Proc. LOPSTR '11*, LNCS, 2012. To appear. Available from [1].

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

[43] H. Zantema. Termination. In Terese, editor, *Term Rewriting Systems*, pages 181–259. Cambridge University Press, 2003.