Termination Analysis of Logic Programs with Cut Using Dependency Triples*

Thomas Ströder, Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Germany
{stroeder,giesl}@informatik.rwth-aachen.de
Peter Schneider-Kamp
IMADA, University of Southern Denmark, Denmark
petersk@imada.sdu.dk

Abstract

In very recent work, we introduced a non-termination preserving transformation from logic programs with cut to definite logic programs. In this paper we extend the transformation such that logic programs with cut are transformed into dependency triple problems instead of definite logic programs. By the implementation of our new method and extensive experiments, we show that the power of automated termination analysis for logic programs with cut is increased substantially.

1 Introduction

Automated termination analysis for logic programs has been widely studied, see, e.g., [3, 4, 5, 11, 13, 14, 17]. Still, virtually all existing techniques only prove universal termination of *definite* logic programs, which do not use the **cut** "!", while most realistic Prolog programs do so. In [16] we introduced a non-termination preserving automated transformation from logic programs with cut to definite logic programs. The transformation consists of two stages. In the first stage we construct a so-called *termination graph* for a given logic program with cut. The second stage is the generation of a definite logic program from this termination graph. In this paper, we improve the second stage of the transformation by generating dependency triple problems instead of definite logic programs from termination graphs.

Dependency triples were introduced in [12] and improved further to the so-called *dependency triple framework* in [15]. Here, the idea was to adapt the successful dependency pair framework [2, 8, 9, 10] from term rewriting to (definite) logic programming. The experiments in [15] showed that this leads to the most powerful approach for automated termination analysis of definite logic programs so far. Our aim is to benefit from this work by providing an immediate translation from termination graphs to dependency triple problems in order to obtain an analysis that preserves termination in more cases.

$$
\begin{array}{rcll}
\mathsf{f}(0,Y) & \leftarrow & !,\mathsf{eq}(Y,0). & (1)\\
\mathsf{f}(X,Y) & \leftarrow & \mathsf{p}(X,P),\mathsf{f}(P,U),\mathsf{f}(U,Y). & (2)\\
\mathsf{p}(0,0). & & & (3)\\
\mathsf{p}(\mathsf{s}(X),X). & & & (4)\\
\mathsf{eq}(X,X). & & & (5)
\end{array}
$$

Figure 1: Example program

Example 1. *To illustrate the concepts and the contributions of this paper, we use the leading example of Fig. 1. It formulates a simplified variant of a functional program from [7, 20] with nested recursion as a logic program. The auxiliary predicate* p *is used to compute the predecessor of a natural number while* eq *is used to unify two terms. See, e.g., [1] for the basics of logic programming.*

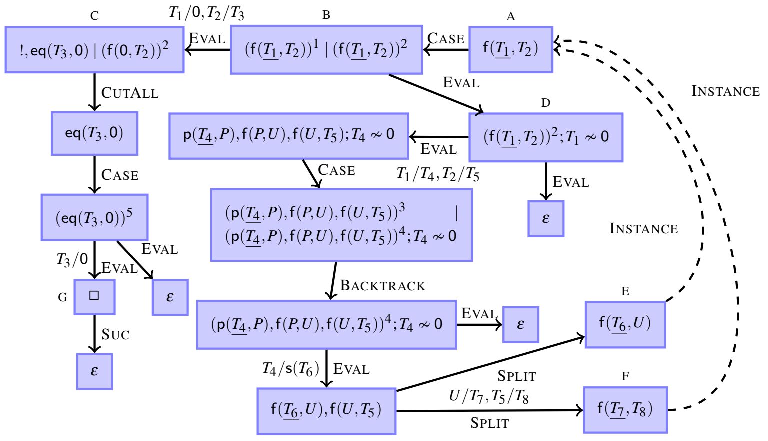
Note that when ignoring cuts, this logic program is not terminating for the set of queries $\mathcal{Q} = \{\mathsf{f}(t_1,t_2) \mid t_1 \textit{ is ground}\}$. *On the other hand, the program terminates if the cuts are taken into account.*

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

In this paper, we first present a termination graph obtained for this example program in Sect. 2 before we apply our new transformation from termination graphs to dependency triple problems in Sect. 3. We show that this new transformation has significant practical advantages in Sect. 4 and, finally, we conclude in Sect. 5.

2 Termination Graphs

Using the method from [16] we obtain the following termination graph for Ex. 1, where we applied some simplifications to ease presentation. The states of this graph contain sequences of *abstract queries*. Here, the *abstract variables* T_i stand for arbitrary terms, whereas underlined abstract variables only stand for ground terms. A sequence of queries $Q_1 \mid Q_2 \mid Q_3 \mid \ldots$ represents the current goal Q_1 and the remaining backtracking possibilities $Q_2, Q_3, \ldots$ in the order of their execution. Sometimes we annotate states by *unification information* such as "$T_4 \not\sim 0$" meaning that T_4 only stands for terms that do not unify with 0. We start in Node A with the state $\mathsf{f}(\underline{T_1}, T_2)$ representing the set of queries $\mathcal{Q}$. Then the termination graph is constructed by a symbolic evaluation of the program. The CASE rule performs Prolog's clause selection rule by labeling the queries with the numbers of the program clauses to indicate which clause to apply next to a query. We applied this rule to the initial node A leading to a node B with two labeled copies of this query. They correspond to the two possibly applicable clauses (1) and (2) in the program. The EVAL rule then performs the resolution with Clause (1), leading to Node C. Moreover, it also produces the child node D which represents those cases where T_1 stands for a term that does not unify with 0. Here, we have to backtrack by removing the first goal from the current state. If we detect that the current goal cannot unify with the head of the corresponding program clause, we use the BACKTRACK rule which is equivalent to the second successor of the EVAL rule. The CUTALL rule drops further backtracking possibilities while the SUC rule backtracks after a successful evaluation, since we examine universal termination. Finally, the SPLIT rule separates two atoms in one query and the INSTANCE rule refers back to a state representing a superset of terms compared to the current state. The SPLIT and INSTANCE rules are needed to obtain a finite graph instead of an infinite tree. We refer to [16] for further details and explanations. In our example, the termination graph of Fig. 2 represents all possible derivations of the program for the set of queries $\mathcal{Q}$ from Ex. 1.



C
$T_1/0, T_2/T_3$
B
A
$!, \mathsf{eq}(T_3, 0) \mid (\mathsf{f}(0, T_2))^2$
EVAL
$(\mathsf{f}(\underline{T_1}, T_2))^1 \mid (\mathsf{f}(\underline{T_1}, T_2))^2$
CASE
$\mathsf{f}(\underline{T_1}, T_2)$
INSTANCE
EVAL
D
CUTALL
$\mathsf{eq}(T_3, 0)$
$\mathsf{p}(\underline{T_4}, P), \mathsf{f}(P, U), \mathsf{f}(U, T_5); T_4 \not\sim 0$
EVAL
$(\mathsf{f}(\underline{T_1}, T_2))^2; T_1 \not\sim 0$
CASE
CASE
$T_1/T_4, T_2/T_5$
EVAL
$(\mathsf{eq}(T_3, 0))^5$
$(\mathsf{p}(\underline{T_4}, P), \mathsf{f}(P, U), \mathsf{f}(U, T_5))^3 \mid$
$(\mathsf{p}(\underline{T_4}, P), \mathsf{f}(P, U), \mathsf{f}(U, T_5))^4; T_4 \not\sim 0$
ε
INSTANCE
$T_3/0$
EVAL
EVAL
BACKTRACK
G
$\square$
ε
$(\mathsf{p}(\underline{T_4}, P), \mathsf{f}(P, U), \mathsf{f}(U, T_5))^4; T_4 \not\sim 0$
EVAL
ε
E
$\mathsf{f}(\underline{T_6}, U)$
SUC
$T_4/\mathsf{s}(T_6)$
EVAL
SPLIT
F
ε
$U/T_7, T_5/T_8$
$\mathsf{f}(\underline{T_6}, U), \mathsf{f}(U, T_5)$
$\mathsf{f}(\underline{T_7}, T_8)$
SPLIT

Figure 2: Termination Graph for Ex. 1.

3 Transformation into Dependency Triple Problems

To prove that all derivations of the example program and the set of queries $\mathcal{Q}$ are terminating, we have to show that the cycles in the termination graph from Fig. 2 cannot be traversed infinitely often when following a derivation of the original program. To this end, we synthesize a dependency triple problem [15] simulating this traversal.

The basic structure in the dependency triple framework is very similar to a clause in logic programming. A *dependency triple* (DT) [12] is a clause $H \leftarrow I, B$ where H and B are atoms and I is a sequence of atoms. Intuitively, such a DT states that a call that is an instance of H can be followed by a call that is an instance of B if the corresponding instance of I can be proven.

Here, a "derivation" is defined in terms of a chain. Let $\mathscr{D}$ be a set of DTs, $\mathscr{P}$ be the program under consideration, and $\mathscr{Q}$ be the class of queries to be analyzed.[1] A (possibly infinite) sequence $(H_0 \leftarrow I_0, B_0), (H_1 \leftarrow I_1, B_1), \ldots$ of variants from $\mathscr{D}$ is a $(\mathscr{D}, \mathscr{Q}, \mathscr{P})$*-chain* iff there are substitutions θ_i, σ_i and an $A \in \mathscr{Q}$ such that $\theta_0 = mgu(A, H_0)$ and for all i, σ_i is an answer substitution for the query $I_i\theta_i$ in the program $\mathscr{P}$, and $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$. Such a tuple $(\mathscr{D}, \mathscr{Q}, \mathscr{P})$ is called a *dependency triple problem* and it is *terminating* iff there is no infinite $(\mathscr{D}, \mathscr{Q}, \mathscr{P})$-chain.

As an example, consider the DT problem $(\mathscr{D}, \mathscr{Q}, \mathscr{P})$ with $\mathscr{D} = \{d_1\}$ where $d_1 = \mathsf{p}(\mathsf{s}(X), Y) \leftarrow \mathsf{eq}(X, Z), \mathsf{p}(Z, Y)$, $\mathscr{Q} = \{\mathsf{p}(t_1, t_2) \mid t_1 \text{ is ground}\}$, and $\mathscr{P} = \{\mathsf{eq}(X, X)\}$. Now, "$d_1$, d_1" is a $(\mathscr{D}, \mathscr{Q}, \mathscr{P})$-chain. To see this, assume that $A = \mathsf{p}(\mathsf{s}(\mathsf{s}(0)), 0)$. Then $\theta_0 = \{X/\mathsf{s}(0), Y/0\}$, $\sigma_0 = \{Z/\mathsf{s}(0)\}$, and $\theta_1 = \{X/0, Y/0\}$.

The basic idea how to synthesize DT problems from termination graphs is to generate DTs for every *triple path* in the termination graph. These are paths which traverse cycles or which connect cycles to the initial state. In Fig. 2 there are two cycles already containing the initial state. As cycles must contain at least one INSTANCE edge, it is sufficient to consider triple paths from successor states of INSTANCE nodes or the initial state to INSTANCE nodes or their successors. So in our example, we have two triple paths: from A to E and from A to F. We use distinct predicate symbols for every state having all distinct variables occurring in the respective state as arguments. The only exception are INSTANCE nodes. Here we use the same predicate symbol as for the successor of the INSTANCE node. So if q is the new predicate symbol for Node A, then A is converted to the atom $\mathsf{q}(T_1, T_2)$, E is converted to $\mathsf{q}(T_6, U)$, and F is converted to $\mathsf{q}(T_7, T_8)$. To transform triple paths into DTs, we use the first node (e.g., A) as the head of the DT and the last node (e.g., E or F) as the last atom of the DT. Moreover, we apply the substitutions on the path to the head of the DT. For the path from A to E we obtain the substitution $[T_1/T_4, T_2/T_5] \circ [T_4/\mathsf{s}(T_6)]$ and, thus, the DT $\mathsf{q}(\mathsf{s}(T_6), T_5) \leftarrow \mathsf{q}(T_6, U)$.

Paths traversing the second successor of a SPLIT node may only be followed if the evaluation of the first SPLIT successor succeeds. This corresponds to the standard goal selection rule. Therefore, we add intermediate goals to the DTs. These goals correspond to the evaluation of first SPLIT successors whenever we traverse a second SPLIT successor. However, for intermediate goals we use different predicate symbols than the ones we used for the head and last body goal of the DTs. For the path from A to F we then obtain the DT $\mathsf{q}(\mathsf{s}(T_6), T_8) \leftarrow \mathsf{r}(T_6, T_7), \mathsf{q}(T_7, T_8)$ using r as the predicate for the intermediate goals.

Now, for the evaluation of intermediate goals we must additionally consider so-called *program paths*. These are paths from successors of INSTANCE nodes or first successors of SPLIT nodes to SUC nodes, INSTANCE nodes, or successors of INSTANCE nodes. However, we can exclude paths traversing other first successors of SPLIT nodes as we are interested in successful evaluations only. In Fig. 2 we have two program paths: from A to F and from A to G. For these paths we generate clauses in the same way as for the DTs with the only exception that we only take the predicate symbol r used for intermediate goals. For SUC nodes, however, we have no body goal and generate facts.

The set of queries for the resulting DT problem contains all queries for the predicate corresponding to the initial state where those positions are assumed to be ground whose corresponding variable is known to represent ground terms in the initial state.

Thus, we obtain the DT problem $(\mathscr{D}_G, \mathscr{Q}_G, \mathscr{P}_G)$ from Fig. 3 for the termination graph G of Fig. 2. This DT problem is easily shown to be terminating by our automated termination prover AProVE.

[1] For simplicity, we use a set of initial queries instead of a general call set as in [15].

$$\begin{array}{rcl} \mathscr{D}_G = \{\mathsf{q}(\mathsf{s}(T_6),T_5) & \leftarrow & \mathsf{q}(T_6,U). \\ \mathsf{q}(\mathsf{s}(T_6),T_8) & \leftarrow & \mathsf{r}(T_6,T_7),\mathsf{q}(T_7,T_8).\} \\ \mathscr{P}_G = \{\mathsf{r}(\mathsf{s}(T_6),T_8) & \leftarrow & \mathsf{r}(T_6,T_7),\mathsf{r}(T_7,T_8). \\ \mathsf{r}(0,0).\} & & \end{array}$$

$\mathscr{Q}_G$ contains all queries $\mathsf{q}(t_1,t_2)$ where t_1 is a ground term.

Figure 3: DT problem for Ex. 1

We now state the central theorem of this paper where we prove that termination of the resulting DT problem implies termination of the original logic program with cut for the set of queries represented by the root state of the termination graph. For the proof we refer to [6].

Theorem 2 (Correctness)**.** *If G is a termination graph for a logic program $\mathscr{P}$ and a set of queries $\mathscr{Q}$ such that the DT problem for G is terminating, then $\mathscr{P}$ is terminating w.r.t. $\mathscr{Q}$.*

Note that the converse of this theorem does not hold.

4 Implementation and Experiments

We implemented the new transformation in our fully automated termination prover AProVE and tested it on all 402 examples for logic programs from the Termination Problem Data Base [19] used for the annual international Termination Competition [18]. We compared the implementation of the new transformation (AProVE DT) with the implementation of the previous transformation into definite logic programs from [16] (AProVE Cut), and with a direct transformation into term rewrite systems ignoring cuts (AProVE Direct) from [14]. We ran the different versions of AProVE on a 2.67 GHz Intel Core i7 and, as in the international Termination Competition, we used a time-out of 60 seconds for each example. For all versions we give the number of examples which could be proved terminating (denoted "Successes"), the number of examples where termination could not be shown ("Failures"), the number of examples for which the timeout of 60 seconds was reached ("Timeouts"), and the total runtime ("Total") in seconds. All details of this empirical evaluation can also be seen online and one can run AProVE on arbitrary examples via a web interface [6].

	AProVE Direct	AProVE Cut	AProVE DT
Successes	243	259	**315**
Failures	144	129	**77**
Timeouts	15	14	**10**
Total	2485.7	3288.0	**2311.6**

Table 1: Experimental results on the TPDB

As shown in Table 1, the new transformation significantly increases the number of examples that can be proved terminating. In particular, we obtain 56 additional proofs of termination compared to [16]. And indeed, for all examples where AProVE Cut succeeds, AProVE DT succeeds, too.

In addition to being more powerful, the new version using dependency triples is also more efficient than any of the two other versions, resulting in fewer timeouts and a total runtime that is less than the one of the direct version and only 70% of the version corresponding to [16].

5 Conclusion

We have shown that the termination graphs introduced in [16] can be used to obtain a transformation from logic programs with cut to dependency triple problems. Our experiments show that this new approach is both considerably more powerful and more efficient than a translation to definite logic programs as in [16]. As the dependency triple framework allows a modular and flexible combination of arbitrary termination techniques from logic programming and even term rewriting, the new transformation to dependency triples can be used as a frontend to any termination tool for logic programs (by taking the union of $\mathscr{D}_G$ and $\mathscr{P}_G$ in the resulting DT problem $(\mathscr{D}_G, \mathscr{Q}_G, \mathscr{P}_G)$) or term rewriting (by using the transformation of [15]).

References

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

[2] T. Arts and J. Giesl. Termination of Term Rewriting using Dependency Pairs. *Theoretical Computer Science*, 236(1,2):133–178, 2000.

[3] M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination Analysis of Logic Programs through Combination of Type-Based Norms. *ACM Transactions on Programming Languages and Systems*, 29(2):Article 10, 2007.

[4] M. Codish, V. Lagoon, and P. J. Stuckey. Testing for Termination with Monotonicity Constraints. In *ICLP '05*, volume 3668 of *LNCS*, pages 326–340, 2005.

[5] D. De Schreye and S. Decorte. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming*, 19,20:199–260, 1994.

[6] Empirical evaluation and proofs for "Termination Analysis of Logic Programs with Cut Using Dependency Triples". `http://aprove.informatik.rwth-aachen.de/eval/cutTriples/`.

[7] J. Giesl. Termination of Nested and Mutually Recursive Algorithms. *Journal of Automated Reasoning*, 19:1–29, 1997.

[8] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *LPAR '04*, volume 3452 of *LNAI*, pages 301–331, 2005.

[9] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[10] N. Hirokawa and A. Middeldorp. Automating the Dependency Pair Method. *Information and Computation*, 199(1,2):172–199, 2005.

[11] F. Mesnard and A. Serebrenik. Recurrence with Affine Level Mappings is P-Time Decidable for CLP(R). *Theory and Practice of Logic Programming*, 8(1):111–119, 2007.

[12] M. T. Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. Termination Analysis of Logic Programs based on Dependency Graphs. In *LOPSTR '07*, volume 4915 of *LNCS*, pages 8–22, 2008.

[13] M. T. Nguyen, D. De Schreye, J. Giesl, and P. Schneider-Kamp. Polytool: Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. *Theory and Practice of Logic Programming*, 2010. To appear.

[14] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting. *ACM Transactions on Computational Logic*, 10(1):2:1–49, 2009.

[15] P. Schneider-Kamp, J. Giesl, and M. T. Nguyen. The Dependency Triple Framework for Termination of Logic Programs. In *LOPSTR '09*, LNCS, 2010. To appear. Preliminary version and experimental details available from `http://aprove.informatik.rwth-aachen.de/eval/PolyAproVE/`.

[16] P. Schneider-Kamp, J. Giesl, T. Ströder, A. Serebrenik, and R. Thiemann. Automated Termination Analysis for Logic Programs with Cut. In *ICLP '10*, 2010. To appear. Preliminary version and experimental details available from `http://aprove.informatik.rwth-aachen.de/eval/cut/`.

[17] A. Serebrenik and D. De Schreye. On Termination of Meta-Programs. *Theory and Practice of Logic Programming*, 5(3):355–390, 2005.

[18] The Termination Competition. `http://www.termination-portal.org/wiki/Termination_Competition`.

[19] The Termination Problem Data Base 7.0 (December 11, 2009). `http://termcomp.uibk.ac.at/status/downloads/`.

[20] C. Walther. On Proving the Termination of Algorithms by Machine. *Artificial Intelligence*, 71(1):101–157, 1994.