The Dependency Triple Framework for Termination of Logic Programs*

Peter Schneider-Kamp[1], Jürgen Giesl[2], and Manh Thang Nguyen[3]

[1] IMADA, University of Southern Denmark, Denmark
[2] LuFG Informatik 2, RWTH Aachen University, Germany
[3] Department of Computer Science, K.U. Leuven, Belgium

Abstract. We show how to combine the two most powerful approaches for automated termination analysis of logic programs (LPs): the *direct* approach which operates directly on LPs and the *transformational* approach which transforms LPs to term rewrite systems (TRSs) and tries to prove termination of the resulting TRSs. To this end, we adapt the well-known *dependency pair framework* from TRSs to LPs. With the resulting method, one can combine arbitrary termination techniques for LPs in a completely modular way and one can use both direct and transformational techniques for different parts of the same LP.

1 Introduction

When comparing the direct and the transformational approach for termination of LPs, there are the following advantages and disadvantages. The *direct* approach is more efficient (since it avoids the transformation to TRSs) and in addition to the TRS techniques that have been adapted to LPs [13,15], it can also use numerous other techniques that are specific to LPs. The *transformational* approach has the advantage that it can use *all* existing termination techniques for TRSs, not just the ones that have already been adapted to LPs.

Two of the leading tools for termination of LPs are Polytool [14] (implementing the direct approach and including the adapted TRS techniques from [13,15]) and AProVE [7] (implementing the transformational approach of [17]). In the annual *International Termination Competition*,[1] AProVE was the most powerful tool for termination analysis of LPs (it solved 246 out of 349 examples), but Polytool obtained a close second place (solving 238 examples). Nevertheless, there are several examples where one tool succeeds, whereas the other does not.

This shows that both the direct and the transformational approach have their benefits. Thus, one should combine these approaches *in a modular way*. In other words, for one and the same LP, it should be possible to prove termination of some parts with the direct approach and of other parts with the transformational

* Supported by *FWO/2006/09: Termination analysis: Crossing paradigm borders* and by the *Deutsche Forschungsgemeinschaft (DFG)*, grant *GI 274/5-2*.

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

D. De Schreye (Ed.): LOPSTR 2009, LNCS 6037, pp. 37–51, 2010.

approach. The resulting method would improve over both approaches and can also prove termination of LPs that cannot be handled by one approach alone.

In this paper, we solve that problem. We build upon [15], where the well-known *dependency pair* (DP) method from term rewriting [2] was adapted in order to apply it to LPs directly. However, [15] only adapted the most basic parts of the method and moreover, it only adapted the classical variant of the DP method instead of the more powerful recent *DP framework* [6,8,9] which can combine different TRS termination techniques in a completely flexible way.

After providing the necessary preliminaries on LPs in Sect. 2, in Sect. 3 we adapt the DP framework to the LP setting which results in the new *dependency triple (DT) framework*. Compared to [15], the advantage is that now arbitrary termination techniques based on DTs can be applied in any combination and any order. In Sect. 4, we present three termination techniques within the DT framework. In particular, we also develop a new technique which can transform *parts* of the original LP termination problem into TRS termination problems. Then one can apply TRS techniques and tools to solve these subproblems.

We implemented our contributions in the tool Polytool and coupled it with AProVE which is called on those subproblems which were converted to TRSs. Our experimental evaluation in Sect. 5 shows that this combination clearly improves over both Polytool or AProVE alone, both concerning efficiency and power.

2 Preliminaries on Logic Programming

We briefly recapitulate needed notations. More details on logic programming can be found in [1], for example. A *signature* is a pair (Σ, Δ) where Σ and Δ are finite sets of function and predicate symbols and $\mathcal{T}(\Sigma, \mathcal{V})$ resp. $\mathcal{A}(\Sigma, \Delta, \mathcal{V})$ denote the sets of all terms resp. atoms over the signature (Σ, Δ) and the variables $\mathcal{V}$. We always assume that Σ contains at least one constant of arity 0. A *clause* c is a formula $H \leftarrow B_1, \ldots, B_k$ with $k \geq 0$ and $H, B_i \in \mathcal{A}(\Sigma, \Delta, \mathcal{V})$. A finite set of clauses $\mathcal{P}$ is a (definite) *logic program*. A clause with empty body is a *fact* and a clause with empty head is a *query*. We usually omit "$\leftarrow$" in queries and just write "$B_1, \ldots, B_k$". The empty query is denoted $\Box$.

For a *substitution* $\delta : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$, we often write $t\delta$ instead of $\delta(t)$, where t can be any expression (e.g., a term, atom, clause, etc.). If δ is a variable renaming (i.e., a one-to-one correspondence on $\mathcal{V}$), then $t\delta$ is a *variant* of t. We write $\delta\sigma$ to denote that the application of δ is followed by the application of σ. A substitution δ is a *unifier* of two expressions s and t iff $s\delta = t\delta$. To simplify the presentation, in this paper we restrict ourselves to ordinary unification with occur check. We call δ the *most general unifier (mgu)* of s and t iff δ is a unifier of s and t and for all unifiers σ of s and t, there is a substitution μ such that $\sigma = \delta\mu$.

Let Q be a query $A_1, \ldots, A_m$, let c be a clause $H \leftarrow B_1, \ldots, B_k$. Then Q' is a *resolvent* of Q and c using δ (denoted $Q \vdash_{c,\delta} Q'$) if $\delta = mgu(A_1, H)$, and $Q' = (B_1, \ldots, B_k, A_2, \ldots, A_m)\delta$. A *derivation* of a program $\mathcal{P}$ and a query Q is a possibly infinite sequence $Q_0, Q_1, \ldots$ of queries with $Q_0 = Q$ where for all i, we have $Q_i \vdash_{c_i,\delta_i} Q_{i+1}$ for some substitution δ_i and some renamed-apart variant c_i of