Termination of Logic Programs via Labelled Term Rewrite Systems

T. Arts and H. Zantema

UU-CS-1994-20
May 1994

Termination of Logic Programs via Labelled Term Rewrite Systems

T. Arts and H. Zantema

Technical Report UU-CS-1994-20
May 1994

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924–3275

Termination of logic programs via labelled term rewrite systems

Thomas Arts and Hans Zantema
Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands.
e-mail: thomas@cs.ruu.nl

Abstract

We propose automatically proving termination of logic programs by transforming them into term rewrite systems (TRS). We describe such a transformation for which termination of the logic program follows from innermost termination of the TRS, which is stronger than previous results. Semantic labelling turns out to be a powerful tool for proving termination of this kind of TRSs: we use it to prove termination of the TRS corresponding to any structural recursive logic program, and also for proving termination of implementations of quick-sort and generation of permutations.

Introduction

There are several approaches to prove termination of logic programs. One of the approaches, introduced by M.R.K. Krishna Rao *et al.* [KKS91], is to transform the logic program into a term rewrite system (TRS) such that the termination property is preserved. More precisely, if the TRS terminates, then the original logic program terminates too. As a consequence, techniques to prove termination of TRSs can be used to prove termination of logic programs. Analyzing termination of a TRS is much more basic than analyzing termination of a logic program, since it does not depend on a particular computation rule and no unification is involved. For TRSs powerful techniques to prove termination automatically are available. One of these techniques is called RPO (recursive path order).

As in [KKS91, GW92, CR, AM93] we propose the following approach for automatically proving termination of logic programs: transform the logic program into a TRS and prove termination of the TRS by existing techniques like RPO and by techniques to be developed. Our main result states that this approach indeed covers a great and important class of logic programs called structural recursive logic programs. But it covers far more logic programs. For example, termination of the logic program

$$\begin{array}{lcl} lesseq(x,x). & & \\ lesseq(s(x),y) & \leftarrow & lesseq(x,y). \\ p(0). & & \\ p(s(x)) & \leftarrow & lesseq(x,y), p(y). \end{array}$$

is not easily seen directly without semantical arguments. But by our transformation it transforms into a TRS of which termination is automatically checked by RPO.

As in [KKS91, GW92, CR, AM93] our transformation is only defined on well-moded logic programs. The first transformation was in [KKS91]. It was improved by the transformation in [GW92] which is able to prove terminating of more logic programs. This algorithm of [GW92] transforms logic programs into conditional rewrite systems. In [CR] a two-step transformation was presented, which extended the transformation in [GW92] with a second translation of the resulting conditional rewrite system into a TRS. Our transformation is inspired by this two-step transformation. Independently in [AM93] a transformation was presented to transform a logic program together with a goal into a TRS. We proved that for our transformation innermost termination of the TRS is sufficient to conclude termination of the logic programs, which is stronger than the other results.

All these transformations have in common that the logic program terminates if the TRS terminates, but unfortunately, only for a subclass of the well-moded logic programs the converse holds. The best result was by G. Aguzzi and U. Modigliani in [AM93]; they introduced the notion of input-driven logic programs and proved that an input-driven logic program terminates if and only if the corresponding TRS according to their transformation terminates. We extend the result in [AM93] by proving that structural recursive programs, which is a class of logic programs that is partly disjoint with the class of input-driven logic programs, transform with our algorithm into terminating TRSs. In this proof we make use of proving termination by semantic labelling [Zan93b]. This technique is very promising as a tool for proving termination of logic programs: it yields termination proofs for the TRSs obtained from programs describing quick-sort or yielding all permutations of a given list.

This paper is organized as follows:

In Section 1 we briefly present some basic notions of logic programming and term rewriting used in this paper, including well-modedness and semantic labelling. In Section 2 we present an algorithm to transform logic programs into term rewrite systems. The proof that this algorithm is correct, i.e., termination is preserved under this translation, can be found in Section 3.

In Section 4 we illustrate the algorithm by some examples. The logic programs in these examples are transformed into TRSs and these TRSs are proved terminating. Some modularity results are given in Section 5. In Section 6 we define the notion of *structural recursive logic programs* and prove that for a structural recursive logic program P the TRS R_P, obtained by applying the algorithm of Section 2 on P, terminates if and only if the logic program P terminates.

1. Preliminaries

1.1. Well-moded logic programs

For a good introduction in logic programming we refer to [Llo87]. In this paper we will only use some standard definitions. For reasons of simplicity we mean 'definite logic program' if we write 'logic program', and 'program clause' if we write 'clause'.

Just like in [CR] and [GW92] we will use a fixed computation rule for the SLD-derivations.

1.1. Definition. The *left-to-right computation rule* is a computation rule that always selects the leftmost atom of a goal. An LD-*derivation* of $P \cup \{G\}$ is an SLD-derivation of $P \cup \{G\}$ via the left-to-right computation rule.

1.2. DEFINITION (cf. [Llo87]). Let P be a program and G a goal. The *LD-tree* for $P \cup \{G\}$ is the tree in which each node of is labelled by a (possibly empty) goal, inductively defined by

1. The root node is labelled by G.
2. Let $\leftarrow A_1, A_2, \ldots, A_m$ $(m \geq 1)$ be a label of a node in the tree. Then, for each program clause $A \leftarrow B_1, \ldots, B_k$ in P such that A_1 and A are unifiable with mgu Θ, the node has a child labelled by
$$\leftarrow (B_1, \ldots, B_k, A_2, \ldots, A_m)\Theta.$$
3. Nodes labelled by the empty clause have no children.

Note that this is a slightly modified version of the standard SLD-tree definition, with the difference that we use the left-to-right computation rule. Like the standard SLD-tree, the LD-tree is finitely branching but can be infinite.

1.3. DEFINITION. A *mode* m of an n-ary predicate p is a function from $\{1, \ldots, n\}$ to the set $\{in, out\}$. The set $\{i \mid m(i) = in\}$ is the set of input positions of p and $\{o \mid m(o) = out\}$ is the set of output positions of p. We say that a variable x occurs in an input (output) position of a literal $p(t_1, \ldots, t_n)$ if it occurs in some t_j such that $m(j) = in$ $(m(j) = out)$.

1.4. DEFINITION. Let x be a variable in the clause $A \leftarrow B_1, \ldots, B_k$. The head A is called a *producer (consumer)* of x, if x occurs in an input (output) position of A, conversely, a body literal B_j is called a *consumer (producer)* of x, if x occurs in an input (output) position of B_j.

1.5. DEFINITION. A clause $B_0 \leftarrow B_1, \ldots, B_n$ is called *LR-well-moded*, if every variable x in the clause has a producer B_i $(0 \leq i \leq n)$ and $i < j$ for every consumer B_j $(1 \leq i \leq n)$ of x in the body of the clause. A program is called *LR-well-moded*, if each of its clauses is LR-well-moded. An *LR-well-moded query* is an LR-well-moded clause without a head.

We consider a notion of termination that depends on the evaluation order of the literals in a query. This evaluation order is motivated by the leftmost selection rule used in Prolog.

1.6. DEFINITION. We call a logic program terminating if there is no well-moded query $\leftarrow Q$ such that the LD-resolution tree of this query is infinite.

1.7. EXAMPLE. The well-moded logic program to compute all even numbers less then a hundred given by

$$\begin{array}{lll}
even(0) & & \\
even(s(s(x))) & \leftarrow & even(x) \\
less(0, s(y)) & & \\
less(s(x), s(y)) & \leftarrow & less(x, y) \\
lesshundred(x) & \leftarrow & less(x, s^{100}(0)), even(x)
\end{array}$$

with modings *even(in)*, *less(out, in)* and *lesshundred(out)* is terminating by our definition. But this program does not terminate for the goal $\leftarrow$ *lesshundred(x)* if the rightmost literal of the body is always selected, as is allowed in a general SLD-derivation. Note that the program is not LR-well-moded any more if the two literal in the body of the last clause are interchanged.

Although we assume the modes of the predicates to be given, this does not imply that mode declarations are to be supplied by the programmer. The problem of automatic generation of mode declarations has been studied by many authors, e.g. [DW88].

Without loss of generality we may assume that every predicate has exactly one mode: if a predicate p is used with n different modes in a program, we consider it as being n distinct predicates, each having one fixed mode. This is illustrated in the following example.

1.8. EXAMPLE. Consider the following logic program to compute permutations of a list:

$$\begin{array}{lcl}
append(nil, l, l). & & \\
append(cons(h, l_1), l_2, cons(h, l_3)) & \leftarrow & append(l_1, l_2, l_3). \\
perm(nil, nil). & & \\
perm(l, cons(h, t)) & \leftarrow & append(v, cons(h, u), l), \\
 & & append(v, u, w), perm(w, t).
\end{array}$$

where *append* has different modes, viz. *append(out, out, in)* and *append(in, in, out)*, such that the last clause of the program is not well-moded. However, this program is well-moded if we choose different predicate symbols for every different moding.

$$\begin{array}{lcl}
append_1(nil, l, l). & & \\
append_1(cons(h, l_1), l_2, cons(h, l_3)) & \leftarrow & append_1(l_1, l_2, l_3). \\
append_2(nil, l, l). & & \\
append_2(cons(h, l_1), l_2, cons(h, l_3)) & \leftarrow & append_2(l_1, l_2, l_3). \\
 & & \\
perm(nil, nil). & & \\
perm(l, cons(h, t)) & \leftarrow & append_2(v, cons(h, u), l), \\
 & & append_1(v, u, w), perm(w, t).
\end{array}$$

1.2. Term rewrite systems

In this section we summarize some preliminaries from term rewriting that we need in this paper.

1.9. DEFINITION. A *signature* is a set $\mathcal{F}$ of *function symbols*. Associated with every $f \in \mathcal{F}$ is a natural number denoting its arity, i.e., the number of arguments it is supposed to have. The function symbols of arity 0 are called *constants*.

1.10. DEFINITION. Let $\mathcal{F}$ be a signature and $\mathcal{V}$ a set of *variables* disjoint from $\mathcal{F}$. The set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of *terms* built from $\mathcal{F}$ and $\mathcal{V}$ is the smallest set with the following two properties:

(i) every variable is a term,

(ii) if $f \in \mathcal{F}$ is an n-ary function symbol and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.
If c is a constant then we write c to denote the term $c()$. For a every term t, $Var(t)$ denotes the set of all variables occurring in t.

1.11. DEFINITION. A *rewrite rule* or *reduction rule* is a pair (l, r) of terms satisfying the following two constrains:
(i) the left-hand side l is not a variable,
(ii) the variables which occur in the right-hand side r also occur in l.
Rewrite rules (l, r) will henceforth be written as $l \to r$.

1.12. DEFINITION. A *term rewrite system* (TRS) is a pair $(\mathcal{F}, \mathcal{R})$ consisting of a signature $\mathcal{F}$ and a set $\mathcal{R}$ of rewrite rules between terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. A TRS is called *finite* if both $\mathcal{F}$ and $\mathcal{R}$ are finite.

1.13. DEFINITION. The rewrite rules of a TRS $(\mathcal{F}, \mathcal{R})$ inductively define a *rewrite relation* $\to_R$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ by
(i) If $l \to r$ is a rewrite rule, then $l^\sigma \to_R r^\sigma$ for every substitution σ,
(ii) If $f \in \mathcal{F}$ is a function symbol with arity n and $t_1, \ldots, t_n$ and t_k' are terms such that $t_k \to_R t_k'$, then $f(t_1, \ldots, t_k, \ldots, t_n) \to_R f(t_1, \ldots, t_k', \ldots, t_n)$.
A term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is called a *normal form* if there is no term $s \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $t \to_R s$. The rewrite rules of a TRS $(\mathcal{F}, \mathcal{R})$ also inductively define a *innermost rewrite relation* $\to_{IN}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ by
(i) If $l \to r$ is a rewrite rule, then $l^\sigma \to_{IN} r^\sigma$ for every substitution σ such that all subterms of l^σ are normal forms,
(ii) If $f \in \mathcal{F}$ is a function symbol with arity n and $t_1, \ldots, t_n$ and t_k' are terms such that $t_k \to_{IN} t_k'$, then $f(t_1, \ldots, t_k, \ldots, t_n) \to_{IN} f(t_1, \ldots, t_k', \ldots, t_n)$.

1.14. DEFINITION. A TRS R is called *terminating* if there exist no infinite reduction of the rewrite relation $\to_R$.

A TRS R is called *innermost terminating* if there exist no infinite reduction of the innermost rewrite relation $\to_{IN}$.

1.15. DEFINITION. For a set $\mathcal{F}$ of operation symbols we define *Emb*$(\mathcal{F})$ to be the TRS consisting of all the rules

$$f(x_1, \ldots, x_n) \to x_i$$

with $f \in \mathcal{F}$ and $i \in \{1, \ldots, n\}$. These rules are called the *embedding rules.*

We can also define a stronger notion of termination called *simple termination.* The definition of simple termination is motivated by [Zan93a].

1.16. DEFINITION. A TRS R over a set $\mathcal{F}$ of function symbols is called *simply terminating* if $R \cup Emb(\mathcal{F})$ is terminating.

A standard technique to prove termination of TRSs, of which several implementations exists, is called RPO (recursive path order). This technique is not applicable to all TRSs. For example it is not applicable to terminating TRSs that are not simply terminating.

1.17. DEFINITION. Let $\triangleright$ be a precedence on a signature $\mathcal{F}$. The *recursive path order* $\succ_{rpo}$ on $\mathcal{T}(\mathcal{F},\mathcal{V})$ is defined as $f(t_1,\ldots,t_n) \succ_{rpo} s$ iff

1. $s = g(s_1,\ldots,s_m)$ and

   (a) $f \triangleright g$ and $f(t_1,\ldots,t_n) \succ_{rpo} s_i$ for all i, $1 \le i \le m$; or

   (b) $f = g$ and $\{t_1,\ldots,t_n\} \succ_{rpo}^{mult} \{s_1,\ldots,s_m\}$,

   where $\succ_{rpo}^{mult}$ is the extension of $\succ_{rpo}$ to multisets, or

2. $t_i \succ_{rpo} s$ or $t_i = s$ for some i, $1 \le i \le n$.

The following theorem has been proved in many articles, among others in [FZ94].

1.18. THEOREM. *Let $(\mathcal{F},\mathcal{R})$ be a TRS and $\triangleright$ a well-founded precedence on the (finite or infinite) signature $\mathcal{F}$. If for all rewrite rules $l \to r$ of $\mathcal{R}$ one has $l \succ_{rpo} r$, then the TRS $(\mathcal{F},\mathcal{R})$ is terminating.*

Constructor systems are a subclass of TRSs. It turns out that the transformations of well-moded logic programs as presented in Section 2 always yields a constructor system.

1.19. DEFINITION (cf. [MT91], [Gra93]). A *constructor system* (CS for short) is a TRS $(\mathcal{F},\mathcal{R})$ with the property that $\mathcal{F}$ can be partitioned into disjoint sets $\mathcal{D}$ and $\mathcal{C}$ such that every left-hand side $F(t_1,\ldots,t_n)$ of a rewrite rule of $\mathcal{R}$ satisfies $F \in \mathcal{D}$ and $t_1,\ldots,t_n \in \mathcal{T}(\mathcal{C},\mathcal{V})$. Function symbols in $\mathcal{D}$ are called *defined symbols* and these in $\mathcal{C}$ *constructors.*

1.3. Semantic labelling

In this section we briefly present a technique to prove termination of TRSs, called *semantic labelling.* For details of this technique we refer to [Zan93b].

In summary, we use semantic labelling as a technique to overcome the deficiency that RPO is only successful on a subclass of simply terminating TRSs, by making RPO applicable to TRSs that are not simply terminating. By this technique a TRS R is transformed into a TRS $\overline{R}$, with the following property:

1.20. PROPOSITION ([Zan93b]). *The TRS R terminates if and only if $\overline{R}$ terminates.*

Therefore, it suffices to prove termination of $\overline{R}$. Our aim is to construct $\overline{R}$ in such a way that it can be proved terminating by RPO.

There is no algorithm that points out how to transform a TRS R into an appropriate TRS $\overline{R}$. To give some intuition on how to use the semantic labelling technique, we explain how we used it in the examples in this paper.

In our examples rewrite rules of the form

$$\begin{array}{rcl} p(\ldots) & \to & k_1(\ldots) \\ k_1(\ldots) & \to & k_2(\ldots) \\ & \vdots & \\ k_n(\ldots) & \to & p(\ldots) \end{array}$$

appear in the TRSs. The problem by using RPO for this situations is that we would like to have the precedence $p \triangleright k_1 \triangleright \ldots \triangleright k_n \triangleright p$, which is impossible. Therefore, the idea is to distinguish between the p on the left-hand side and the p on the right-hand side by some interpretation of its arguments.

1.21. EXAMPLE. The following TRS R can not be proved terminating by RPO, since it is not simply terminating:

$$p(s(x)) \rightarrow p(c(s(x))).$$

We shall show that by semantic labelling this TRS R transforms into the TRS $\overline{R}$

$$p_1(s(x)) \rightarrow p_0(c(s(x))).$$

Note that p in R is replaced by the labelled symbols p_0 and p_1 in $\overline{R}$. We immediately see that $\overline{R}$ terminates by RPO. Now Proposition 1.20 ensures that also R terminates.

Now we describe the semantic labelling technique in detail. Let $(\mathcal{F}, \mathcal{R})$ be the TRS that we want to prove terminating. We define an *$\mathcal{F}$-algebra* $\mathcal{M}$ to consist of a set M (the carrier set, typically the natural numbers or a set of terms) and for every $f \in \mathcal{F}$ of arity n a function $f_{\mathcal{M}} : M^n \rightarrow M$.

In Example 1.21 the carrier set M is chosen to be the set $\{\mathbf{0}, \mathbf{1}\}$ and the functions are chosen as follows: $p_{\mathcal{M}}(x) = \mathbf{0}, s_{\mathcal{M}}(x) = \mathbf{1}, c_{\mathcal{M}}(x) = \mathbf{0}$ for $x \in \{\mathbf{0}, \mathbf{1}\}$. For reasons of simplicity we write p, s and c in stead of $p_{\mathcal{M}}$, $s_{\mathcal{M}}$ and $c_{\mathcal{M}}$ respectively.

The functions in the algebra have to be chosen in such a way that for the interpretation function $[\![]\!]$ from terms and valuations ρ to the carrier set defined by

$$\begin{aligned} [\![x]\!]_\rho &= \rho(x) \\ [\![f(t_1, \dots, t_n)]\!]_\rho &= f_{\mathcal{M}}([\![t_1]\!]_\rho, \dots, [\![t_n]\!]_\rho) \end{aligned}$$

the $\mathcal{F}$-algebra is a *model* for R, i.e., for all valuations ρ and all rewrite rules $l \rightarrow r$ in R

$$[\![l]\!]_\rho = [\![r]\!]_\rho$$

In Example 1.21 $[\![p(s(x))]\!]_\rho = \mathbf{0} = [\![p(c(s(x)))]\!]_\rho$ for all valuations ρ, hence $\mathcal{M}$ is a model for R.

Next we choose for every $f \in \mathcal{F}$ a corresponding non-empty set S_f of labels. This defines the new signature $\overline{\mathcal{F}} = \{f_s | f \in \mathcal{F}, s \in S_f\}$, where the arity of f_s is defined to be the arity of f. A function symbol f is called *labelled* if S_f contains more than one element. For unlabelled f the set S_f, containing only one element, can be left implicit; in that case we write f instead of f_s.

In Example 1.21 we choose $S_p = \{\mathbf{0}, \mathbf{1}\}$ and leave s and c unlabelled, hence $\overline{\mathcal{F}}$ is defined to be the set $\{s, c, p_0, p_1\}$.

We choose for every $f \in \mathcal{F}$ a map $\pi_f : M^n \rightarrow S_f$, where n is the arity of f. This map describes how a function symbol is labelled depending on the value of its arguments as interpreted in the algebra $\mathcal{M}$. For unlabelled f this map π_f can be left implicit. The labelling of function symbols induces a labelling of terms via the function lab from terms and valuations to labelled terms defined by

$$\begin{aligned} \mathsf{lab}(x, \rho) &= x \\ \mathsf{lab}(f(t_1, \dots, t_n), \rho) &= f_{\pi_f([\![t_1]\!]_\rho, \dots, [\![t_n]\!]_\rho)}(\mathsf{lab}(t_1, \rho), \dots, \mathsf{lab}(t_n, \rho)) \end{aligned}$$

In Example 1.21 π_p is chosen to be the identity function. Intuitively this means that p is labelled by the interpretation of its argument. Note that we have $[\![s(x))]\!]_\rho = \mathbf{1}$ and $[\![c(s(x)))]\!]_\rho = \mathbf{0}$ for all valuations ρ. Thus,

$$\mathsf{lab}(p(s(x)), \rho) = p_{\pi_p([\![s(x)]\!]_\rho)}(\mathsf{lab}(s(x), \rho)) = p_1(s(x))$$

and

$$\mathsf{lab}(p(c(s(x))),\rho) = p_{\pi_p([\![c(s(x))]\!]_\rho)}(\mathsf{lab}(c(s(x)),\rho)) = p_{\mathbf{0}}(c(s(x)))$$

for all valuations ρ.

Now we define how to obtain $\overline{R}$ from R. Let R be a TRS over a signature $\mathcal{F}$. Fix an $\mathcal{F}$-algebra $\mathcal{M}$ together with corresponding sets S_f and functions π_f, such that $\mathcal{M}$ is a model for R. The TRS $\overline{R}$ over $\overline{\mathcal{F}}$ is defined as the TRS consisting of the rules

$$\mathsf{lab}(l,\rho) \rightarrow \mathsf{lab}(r,\rho)$$

for all valuations ρ and all rewrite rules $l \rightarrow r$ of R.

In Example 1.21 $\overline{R}$ is defined in this way and since $\overline{R}$ is terminating, Proposition 1.20 ensures termination of R.

1.22. Example. The following TRS is the translation of a logic program and a typical example of the type of TRSs we consider in this paper.

$$\begin{array}{lcl} q(x) & \rightarrow & out_1(s(x)) \\ p(s(x)) & \rightarrow & k_1(x,q(x)) \\ k_1(x,out_1(y)) & \rightarrow & k_2(x,y,p(c(s(x)))) \\ k_2(x,y,out) & \rightarrow & out \end{array}$$

This TRS can not be proved terminating by RPO. Similar to the TRS of Example 1.21, this TRS can be transformed by semantic labelling. The chosen carrier set is $\{\mathbf{0},\mathbf{1}\}$ again. To obtain a model we choose

$$p_{\mathcal{M}}(x) = q_{\mathcal{M}}(x) = out_{1,\mathcal{M}}(x) = k_{1,\mathcal{M}}(x,y) = k_{2,\mathcal{M}}(x,y,z) = out = \mathbf{0}$$

We also choose $s_{\mathcal{M}}(x) = \mathbf{1}$ and $c_{\mathcal{M}}(x) = \mathbf{0}$. Since p is the only symbol that we want to label, we choose $S_p = \{\mathbf{0},\mathbf{1}\}$ and leave all other function symbols unlabelled. The map π_p is chosen to be the identity. The obtained TRS is

$$\begin{array}{lcl} q(x) & \rightarrow & out_1(s(x)) \\ p_{\mathbf{1}}(s(x)) & \rightarrow & k_1(x,q(x)) \\ k_1(x,out_1(y)) & \rightarrow & k_2(x,y,p_{\mathbf{0}}(c(s(x)))) \\ k_2(x,y,out) & \rightarrow & out \end{array}$$

This TRS can easily be proved terminating by RPO.

2. Transforming a logic program into a rewrite system

In this section we present an algorithm to transform well-moded logic programs into term rewrite systems. One of the first successful approaches of such an algorithm was given in [KKS91]. This approach was improved in [GW92], where a more powerful algorithm was presented to transform well-moded logic programs to conditional rewrite systems. In [CR] a two-step translation was presented which extended the transformation of [GW92] with a second translation of the resulting conditional rewrite system into a TRS. Our algorithm is inspired by this two-step translation; the resulting TRSs are essentially the same. The claim in [CR] is that if the TRS obtained by transforming a logic program is simply terminating, then the logic program terminates. We prove the stronger result that if the TRS obtained by transforming a logic program is innermost terminating, then the logic program terminates. In [AM93] an algorithm is considered to transform a well-moded logic program together with a well-moded goal into a TRS and if this TRS obtained by transforming a logic program with a goal is terminating, then the logic program terminates for the given goal. Innermost termination implies termination, hence our result is even stronger than this independent approach of [AM93].

In the algorithm the literals are denoted in such a way that the first arguments are the *in*-positions and the last arguments are the *out*-positions. In the algorithm we write $p(t_1, \ldots, t_k, t_{k+1}, \ldots, t_{k+k'})$ to denote that $t_1, \ldots, t_k$ are the arguments on input positions and $t_{k+1}, \ldots, t_{k+k'}$ are the arguments on output positions.

2.1. DEFINITION. Let $t_1, \ldots, t_n$ be terms. We write $var(t_1, \ldots, t_n)$ for the ordered sequence (with respect to some fixed total order on variables) of all variables in $t_1, \ldots, t_n$. For example $var(p(z), max(x, x, x), less(0, y))$ is the sequence x, y, z.

```
program Transform ($P$:in, $R_P$:out);
begin
    $R_P$:=$\emptyset$; $index$:=0;
    for each clause $B_0 \leftarrow B_1, \ldots, B_m \in P$ do
    begin
        Let $B_0$ be $p_0(t_1, \ldots, t_k, t_{k+1}, \ldots, t_{k+k'})$;
        $Out := out_{k'}(t_{k+1}, \ldots, t_{k+k'})$;
        if ($m = 0$)          /* Program clause without body */
        then $R_P := R_P \cup \{p_0(t_1, \ldots, t_k) \to Out\}$;
        else                  /* Program clause with body */
        begin
            for $j$ = 1 to $m$ do
            begin
                Let $B_{j-1}$ be $p_{j-1}(t_1, \ldots, t_k, t_{k+1}, \ldots, t_{k+k'})$;
                Let $B_j$ be $p_j(s_1, \ldots, s_l, s_{l+1}, \ldots, s_{l+l'})$;
                $index := index + 1$;
                $Varold := Var$;
                $Var := var(t_1, \ldots, t_k)$;
                if ($j = 1$)
                then
                    $R_P := R_P \cup \{p_0(t_1, \ldots, t_k) \to k_{index}(Var, p_j(s_1, \ldots, s_l))\}$;
                else
                    $R_P := R_P \cup \{k_{index-1}(Var, out_{k'}(t_{k+1}, \ldots, t_{k+k'})) \to$
                                 $k_{index}(Varold, p_j(s_1, \ldots, s_l))\}$;
            end
            $R_P := R_P \cup \{k_{index}(Var, out_{k'}(s_{l+1}, \ldots, s_{l+l'})) \to Out\}$;
        end
    end
end
```

As mentioned before, the resulting TRS is always a constructor system (see Section 5). The defined symbols of this constructor system are the predicate symbols and the in the algorithm introduced k-symbols. The constructor symbols are the function symbols and constants of the logic program, also denoted by $\mathcal{F}un$, and the in the algorithm introduced *out*-symbols, also denoted by $\mathcal{O}ut$.

3. Correctness of the transformation

Note that there are no while loops in the transformation algorithm, hence the algorithm terminates.

3.1. LEMMA. *For each rule $l \to r$ in R_P one has $Var(r) \subseteq Var(l)$.*

PROOF. There are two possibilities. The rule $l \to r$ may be the translation of the program clause

- $B_0 = p_0(t_1, \dots, t_k, t_{k+1}, \dots, t_{k+k'})$. Since the clause is well-moded every variable has a producer, hence all variables that occur, occur also in the in-positions. By translating this clause in
$$p_0(t_1, \dots, t_k) \to out_{k'}(t_{k+1}, \dots, t_{k+k'})$$
it is clear that $Var(r) \subseteq Var(l)$.
- $B_0 \leftarrow B_1, \dots, B_n$. The translation is of the form:
$$\begin{array}{rcl} p_0(\dots) & \to & k_1(var, p_1(\dots)) \\ k_1(var, out^{(1)}(\dots)) & \to & k_2(var^{(1)}, p_2(\dots)) \\ & \vdots & \\ k_{n-1}(var^{(n-2)}, out^{(n-1)}(\dots)) & \to & k_n(var^{(n-1)}, p_n(\dots)) \\ k_n(var^{(n-1)}, out^{(n)}(\dots)) & \to & out^{(0)}(\dots) \end{array}$$
Where $out^{(i)}$ stands for out_m if the number of out-positions of B_i is m. Note that $var^{(i)}$ in the right hand side always contains all variables of the left hand side. Hence, $var \subseteq var^{(1)} \subseteq \dots \subseteq var^{(n-1)}$. The clause is well-moded, thus, if there is a variable in the in-position of B_i $(i > 0)$, i.e., in the right hand side
$$k_i(var^{(i-1)}, p_i(\dots))$$
then there is a $j < i$ such that the variable is in the out-position of B_j for $(j > 0)$ or in the in-position of B_0, hence the variable occurs in the left hand side in $var^{(i-2)}$ or in the $out^{(i-1)}$. The variables in the out-positions of B_0 are consumers, hence they occur as producer in the body of the clause, thus in $var^{(n)}$ or $out^{(n)}(\dots)$.

∎

From the transformation algorithm it is clear that the obtained TRS R_P is finite.

3.2. LEMMA. *Let $\leftarrow q(t_1, \dots, t_k, t_{k+1}, \dots, t_{k+k'})$ be a well-moded query to a well-moded program P. If there is an LD-refutation from the query and the logic program P then there is an innermost reduction*
$$q(t_1, \dots, t_k) \to^+_{R_P} out_{k'}(t_{k+1}\Theta, \dots, t_{k+k'}\Theta)$$
for some substitution Θ.

Note that in a well-moded query $q(\ldots)$ there are no variables on the in-positions. Hence $t_1, \ldots, t_k$ are ground. Therefore the terms $t_{k+1}\Theta, \ldots, t_{k+k'}\Theta$ are ground.

In the proof it is very important that the terms on the in-positions are ground and that all the mgu's in the LD-derivation are substitutions of variables in the out-positions.

PROOF. The proof is by induction on the length of the LD-path in the LD-resolution tree.

- If the length of the LD-path is 1, then there exists a program clause

  $$q(s_1, \ldots, s_k, s_{k+1}, \ldots, s_{k+k'})$$

  that resolves with the query by mgu Θ, i.e., $t_j\Theta = s_j\Theta$ for all $j \in \{1, \ldots, k, k+1, \ldots, k+k'\}$. The associated rewrite rule of this program clause is:

  $$q(s_1, \ldots, s_k) \to out_{k'}(s_{k+1}, \ldots, s_{k+k'}).$$

  Because the query is well-moded, all parameters on the in-positions are ground. Hence $t_j = t_j\Theta = s_j\Theta$ for $1 \leq j \leq k$ and

  $$\begin{array}{rcl} q(t_1, \ldots, t_k) & = & \\ q(s_1\Theta, \ldots, s_k\Theta) & \to_{R_P} & out_{k'}(s_{k+1}\Theta, \ldots, s_{k+k'}\Theta). \end{array}$$

  Since Θ is a mgu, we have $s_j\Theta = t_j\Theta$ for $k+1 \leq j \leq k+k'$. The reduction is an innermost reduction, since no predicate symbol, and hence no redex, occurs in the term s_j $(1 \leq j \leq k)$.

- If the length of the LD-path is n+1, then the query resolves with a program clause

  $$q(s_1, \ldots, s_k, s_{k+1}, \ldots, s_{k+k'}) \leftarrow p_1(\ldots), p_2(\ldots), \ldots, p_m(\ldots).$$

  via mgu Θ_0, say. Since the query $q(\ldots)$ has an LD-refutation, all atoms $p_j(\ldots)$ have an LD-refutation of shorter length.

  The associated rewrite rules of this program clause are of the form:

  $$\begin{array}{rcl} q(s_1, \ldots, s_k) & \to & k_1(var, p_1(\ldots)) \\ k_1(var, out^{(1)}(\ldots)) & \to & k_2(var^{(1)}, p_2(\ldots)) \\ & \vdots & \\ k_{m-1}(var^{(m-2)}, out^{(m-1)}(\ldots)) & \to & k_m(var^{(m-1)}, p_m(\ldots)) \\ k_m(var^{(m-1)}, out^{(m)}(\ldots)) & \to & out_{k'}(s_{k+1}, \ldots, s_{k+k'}) \end{array}$$

  Because the query is well-moded, all parameters on the in-positions are ground. Hence, $t_j = t_j\Theta_0 = s_j\Theta_0$ for $1 \leq j \leq k$ are ground and also $var\Theta_0$ is ground.

  $$\begin{array}{rcl} q(t_1, \ldots, t_k) & = & \\ q(s_1\Theta_0, \ldots, s_k\Theta_0) & \to_{R_P} & k_1(var\Theta_0, p_1(\ldots)\Theta_0). \end{array}$$

The terms s_j $(1 \leq j \leq k)$ do not contain a redex, hence the reduction is an innermost reduction. By the induction hypothesis there is an innermost reduction, such that

$$k_1(var\Theta_0, p_1(\ldots)\Theta_0) \quad \rightarrow^{+}_{R_P} \quad k_1(var\Theta_0, out^{(1)}(\ldots)\Theta_0\Theta_1)$$

for some substitution Θ_1. Since $var\Theta_0$ is ground, $var\Theta_0 = var\Theta_0\Theta_1$. Since $\Theta_0\Theta_1$ is a substitution

$$k_1(var\Theta_0\Theta_1, out^{(1)}(\ldots)\Theta_0\Theta_1) \quad \rightarrow_{R_P} \quad k_2(var^{(1)}\Theta_0\Theta_1, p_2(\ldots)\Theta_0\Theta_1)$$

This reduction is an innermost reduction, since $out^{(1)}(\ldots)$ does not contain a redex. Note that also $out^{(1)}(\ldots)\Theta_0\Theta_1$ is ground, thus $var^{(1)}\Theta_0\Theta_1$ is ground. Using those arguments m times we obtain:

$$\begin{array}{rc}
k_2(var^{(1)}\Theta_0\Theta_1, p_2(\ldots)\Theta_0\Theta_1) & \rightarrow_{R_P} \\
k_2(var^{(1)}\Theta_0\Theta_1, out^{(2)}(\ldots)\Theta_0\Theta_1\Theta_2) & \rightarrow_{R_P} \\
k_3(var^{(2)}\Theta_0\Theta_1\Theta_2, p_3(\ldots)\Theta_0\Theta_1\Theta_2) & \rightarrow_{R_P} \\
 & \vdots \\
k_m(var^{(m-1)}\Theta_0\ldots\Theta_{m-2}, p_m(\ldots)\Theta_0\ldots\Theta_{m-2}) & \rightarrow_{R_P} \\
k_m(var^{(m-1)}\Theta_0\ldots\Theta_{m-2}, out^{(m)}(\ldots)\Theta_0\ldots\Theta_{m-1}) & \rightarrow_{R_P} \\
out_{k'}(s_{k+1}, \ldots, s_{k+k'})\Theta & = \\
out_{k'}(t_{k+1}\Theta, \ldots, t_{k+k'}\Theta) &
\end{array}$$

where $\Theta = \Theta_0 \ldots \Theta_m$.

∎

3.3. THEOREM. *Let $\leftarrow q(t_1, \ldots, t_k, t_{k+1}, \ldots, t_{k+k'})$ be a well-moded query to a well-moded program P. If there is an infinite LD-derivation from this query, then there is an infinite innermost reduction in R_P starting with $q(t_1, \ldots, t_k)$.*

PROOF. We prove this by induction on the structure of the infinite LD-tree with root node $\leftarrow q(t_1, \ldots, t_k, t_{k+1}, \ldots, t_{k+k'})$. Since there is an infinite LD-derivation the query resolves with a program clause of the form $q(s_1, \ldots, s_k, s_{k+1}, \ldots, s_{k+k'}) \leftarrow p_1(\ldots), p_2(\ldots), \ldots, p_m(\ldots)$ via mgu Θ_0, say. The associated rewrite rules of this program clause are of the form:

$$\begin{array}{rcl}
q(s_1, \ldots, s_k) & \rightarrow & k_1(var, p_1(\ldots)) \\
k_1(var, out^{(1)}(\ldots)) & \rightarrow & k_2(var^{(1)}, p_2(\ldots)) \\
 & \vdots & \\
k_{m-1}(var^{(m-2)}, out^{(m-1)}(\ldots)) & \rightarrow & k_m(var^{(m-1)}, p_m(\ldots)) \\
k_m(var^{(m-1)}, out^{(m)}(\ldots)) & \rightarrow & out_{k'}(s_{k+1}, \ldots, s_{k+k'})
\end{array}$$

Because the query is well-moded all parameters on the in-positions are ground. Hence $t_j = t_j\Theta_0 = s_j\Theta_0$ for $1 \leq j \leq k$. We have:

$$q(t_1, \ldots, t_k) = q(s_1\Theta_0, \ldots, s_k\Theta_0) \rightarrow_{R_P} k_1(var\Theta_0, p_1(\ldots)\Theta_0).$$

The reduction is an innermost reduction, since no predicate symbol, and hence no redex, occurs in the term s_j $(1 \leq j \leq k)$. Now we have two possibilities:

1. $p_1(\ldots)$ has an infinite LD-derivation. Then by induction we have an infinite innermost rewrite reduction of $p_1(\ldots)\Theta_0$ and since $p_1(\ldots)\Theta_0$ is an innermost redex of $k_1(var\Theta_0, p_1(\ldots)\Theta_0)$ we also have an innermost reduction of $q(t_1, \ldots, t_k)$.

2. $p_1(\ldots)$ has no infinite LD-derivation, hence this query has an LD-refutation. By Lemma 3.2 we know that there is an innermost reduction

$$p_1(\ldots)\Theta_0 \to^+_{R_P} out^{(1)}(\ldots)\Theta_0\Theta_1$$

for some substitution Θ_1. Note also that $var\Theta_0$ is ground and hence $var\Theta_0 = var\Theta_0\Theta_1$. Therefore we have

$$\begin{array}{rl} k_1(var\Theta_0, p_1(\ldots)\Theta_0\Theta_1) & \to^+_{R_P} \; k_1(var\Theta_0, out^{(1)}(\ldots)\Theta_0\Theta_1) \\ & \to_{R_P} \; k_2(var^{(1)}\Theta_0\Theta_1, p_2(\ldots)\Theta_0\Theta_1) \end{array}$$

where all reductions are innermost reductions. By induction we have an infinite innermost rewrite reduction.

∎

3.4. COROLLARY. *Let P be a well-moded logic program R_P the associated TRS. If the TRS R_P is innermost terminating, then the logic program P terminates, i.e. there is no well-moded query $\leftarrow Q$ such that the LD-resolution tree of this query is infinite.*

PROOF. Immediate from Theorem 3.3. ∎

4. Examples of the transformation

The purpose of the following examples is to illustrate the transformation algorithm.

4.1. EXAMPLE. The example from the introduction transforms into the rewrite system

$$\begin{array}{lll} lesseq(x) & \to & out_1(x) \\ lesseq(s(x)) & \to & k_1(x, lesseq(x)) \\ k_1(x, out_1(y)) & \to & out_1(y) \\ p(0) & \to & out \\ p(s(x)) & \to & k_2(x, lesseq(x)) \\ k_2(x, out_1(y)) & \to & k_3(x, y, p(y)) \\ k_3(x, y, out) & \to & out \end{array}$$

As we already mentioned in the introduction this TRS is terminating by RPO based on the precedence $lesseq \rhd k_1 \rhd out_1$ and $s \rhd k_2 \rhd p, k_3 \rhd lesseq \rhd out$.

4.2. EXAMPLE. The one-line logic program $p(s(x)) \leftarrow p(c(s(x)))$ translates into the terminating TRS

$$\begin{array}{lll} p(s(x)) & \to & k_1(x, p(c(s(x)))) \\ k_1(x, out) & \to & out \end{array}$$

Note that this TRS is not simply terminating since by adding the embedding rules we have the infinite reduction sequence

$$p(s(x)) \to k_1(x, p(c(s(x)))) \to_{Emb} k_1(x, p(s(x))) \to \ldots$$

Therefore, RPO can not be used to prove termination of this TRS. By the technique of transformation orderings and the heuristics of [Ste92] this TRS can be proved terminating automatically. Also by using semantic labelling this TRS is easily proved to be terminating, in Section 1.3 the semantic labelling technique is explained with as a leading example a TRS that is very similar to the TRS of this example.

Proving termination of the logic programs in the following three examples goes beyond syntactical analysis. No straightforward method to automatically prove termination of these programs is known. We use the semantical information that there is an argument that decreases (in a certain way) for every recursive call. One way to use the semantical information is semantic labelling.

4.3. EXAMPLE. The logic program

$$\begin{array}{lll} less(0,s(x)). & & \\ less(s(x),s(y)) & \leftarrow & less(x,y). \\ & & \\ max(x,x,x). & & \\ max(x,y,x) & \leftarrow & less(y,x). \\ max(x,y,z) & \leftarrow & less(x,y), max(y,x,z). \end{array}$$

with modings $less(in, in)$ and $max(in, in, out)$ is an inefficient but correct way to compute the maximum of two given numbers. It is hard to prove termination of this logic program. With our method it transforms into the TRS

$$\begin{array}{lll} less(0,s(x)) & \rightarrow & out \\ less(s(x),s(y)) & \rightarrow & k_1(x,y,less(x,y)) \\ k_1(x,y,out) & \rightarrow & out \\ & & \\ max(x,x) & \rightarrow & out_1(x) \\ max(x,y) & \rightarrow & k_2(x,y,less(y,x)) \\ k_2(x,y,out) & \rightarrow & out_1(x) \\ max(x,y) & \rightarrow & k_3(x,y,less(x,y)) \\ k_3(x,y,out) & \rightarrow & k_4(x,y,max(y,x)) \\ k_4(x,y,out_1(z)) & \rightarrow & out_1(z) \end{array}$$

that is not simply terminating and for this TRS a termination proof is still difficult. With semantic labelling the obtained TRS can be transformed to a labelled TRS, which can be proved terminating by RPO. With the technique of semantic labelling we use the information that in a recursive call the first argument of *max* is always greater than the second argument. This results in the choice of an $\mathcal{F}$-algebra $\mathcal{M}$ with carrier set $\mathbb{N}$ and functions

$$\begin{array}{rcl}
0_{\mathcal{M}} & = & 0 \\
s_{\mathcal{M}}(x) & = & x+1 \\
less_{\mathcal{M}}(x,y) & = & \begin{cases} 1 & x<y \\ 0 & x \geq y \end{cases} \\
out_{\mathcal{M}} & = & 1 \\
k_{1,\mathcal{M}}(x,y,z) & = & z \\
max_{\mathcal{M}}(x,y) & = & max(x,y) \\
k_{2,\mathcal{M}}(x,y,z) & = & \begin{cases} x & \text{if } z=1 \\ y & \text{otherwise} \end{cases} \\
k_{3,\mathcal{M}}(x,y,z) & = & max(x,y) \\
k_{4,\mathcal{M}}(x,y,z) & = & z \\
out_{1,\mathcal{M}}(x) & = & x
\end{array}$$

One easily verifies that this algebra is a model for the above TRS. Now we want to replace the symbols max and k_3 with labelled symbols. The motivation to replace not only the symbol max, but also the symbol k_3, is that we want to have some book-keeping from which max symbol the k_3 symbol stems. We choose as sets of labels $S_{max} = \mathbb{N}$ and $S_{k_3} = \mathbb{N} \times \mathbb{N}$. The label functions are chosen as $\pi_{max}(x,y) = less_{\mathcal{M}}(x,y)$ and $\pi_{k_3}(x,y,z) = (less_{\mathcal{M}}(x,y),z)$. Note that, although the sets of labels S_{max} and S_{k_3} are infinite, only finitely may values are assigned to π_{max} and π_{k_3}. Semantic labelling results in the following labelled TRS

$$\begin{array}{lcl}
less(0,s(x)) & \to & out \\
less(s(x),s(y)) & \to & k_1(x,y,less(x,y)) \\
k_1(x,y,out) & \to & out \\
\\
max_0(x,x) & \to & out_1(x) \\
max_0(x,y) & \to & k_2(x,y,less(y,x)) \\
max_1(x,y) & \to & k_2(x,y,less(y,x)) \\
k_2(x,y,out) & \to & out_1(x) \\
max_0(x,y) & \to & k_{3,(0,0)}(x,y,less(x,y)) \\
max_1(x,y) & \to & k_{3,(1,1)}(x,y,less(x,y)) \\
k_{3,(0,1)}(x,y,out) & \to & k_4(x,y,max_1(y,x)) \\
k_{3,(1,1)}(x,y,out) & \to & k_4(x,y,max_0(y,x)) \\
k_4(x,y,out_1(z)) & \to & out_1(z)
\end{array}$$

This TRS can be proved terminating by RPO with precedence $k_{3,(0,1)} \rhd max_1 \rhd k_{3,(1,1)} \rhd max_0 \rhd k_{3,(0,0)} \rhd k2, k4 \rhd less \rhd k_1 \rhd out_1, out$.

4.4. EXAMPLE. The logic program to compute all permutations of a given list as described in example 1.8, is transformed by the algorithm of Section 2 into the TRS:

$$\begin{array}{lcl}
append_1(nil, l) & \to & out_1(l) \\
append_1(cons(h, l_1), l_2) & \to & k_1(h, l_1, l_2, append_1(l_1, l_2)) \\
k_1(h, l_1, l_2, out_1(l_3)) & \to & out_1(cons(h, l_3)) \\
\\
append_2(l) & \to & out_2(nil, l) \\
append_2(cons(h, l_3)) & \to & k_2(h, l_3, append_2(l_3)) \\
k_2(h, l_3, out_2(l_1, l_2)) & \to & out_2(cons(h, l_1), l_2) \\
\\
perm(nil) & \to & out_1(nil) \\
perm(l) & \to & k_3(l, append_2(l)) \\
k_3(l, out_2(v, cons(h, u))) & \to & k_4(l, h, v, u, append_1(v, u)) \\
k_4(l, h, v, u, out_1(w)) & \to & k_5(l, h, v, u, w, perm(w)) \\
k_5(l, h, v, u, w, out_1(t)) & \to & out_1(cons(h, t))
\end{array}$$

Note that we obtain two different translations for the *append* predicate, depending on the two modings of *append* in the program. Semantic labelling succeeds in proving termination of the TRS, using the observation that a recursive call is always applied on a shorter list. To express the length of a list we choose the natural numbers $\mathbb{N}$ as the carrier set of an $\mathcal{F}$-algebra $\mathcal{M}$. The functions for the symbols *nil* and *cons* are chosen in such a way that they represent the length of the list, viz. $nil_{\mathcal{M}} = 0$ and $cons_{\mathcal{M}}(h, t) = 1 + t$. The functions for the predicate symbols, ranging over lists, are chosen to be the length of the lists that are given as input arguments for these predicates. Since the output arguments of these predicates are the same with respect to the length of lists, the functions for the *out* symbols are also easy to choose:

$$\begin{array}{rcl}
append_{1,\mathcal{M}}(x, y) & = & x + y \\
append_{2,\mathcal{M}}(x) & = & x \\
perm_{\mathcal{M}}(x) & = & x \\
out_{1,\mathcal{M}}(x) & = & x \\
out_{2,\mathcal{M}}(x, y) & = & x + y
\end{array}$$

The functions corresponding with the other symbols are chosen in such a way that $\mathcal{M}$ is indeed a model for the above TRS.

$$\begin{array}{rcl}
k_{1,\mathcal{M}}(h, x, y, z) & = & 1 + z \\
k_{2,\mathcal{M}}(h, x, y) & = & 1 + y \\
k_{3,\mathcal{M}}(x, y) & = & y \\
k_{4,\mathcal{M}}(x, h, y, z, w) & = & 1 + w \\
k_{5,\mathcal{M}}(x, h, y, z, v, w) & = & 1 + w
\end{array}$$

Now we want to replace the symbol *perm* by a labelled symbol and, just as in the previous example, we also have to replace k_3 and k_4. As sets of labels we choose $S_{perm} = S_{k_3} =$

$S_{k_4} = \mathbb{N}$, since the intuition is that we label with the length of lists. Led by this intuition we choose the following mappings $\pi_{perm(x)} = x$, $\pi_{k_3}(x,y) = y$ and $\pi_{k_4}(x,h,y,z,w) = 1+w$. This results in the following labelled TRS, where the rules with the predicates *perm*, k_3 and k_4 of the above TRS are replaced by infinitely many rules:

$$\begin{array}{lcl}
append_1(nil,l) & \to & out_1(l) \\
append_1(cons(h,l_1),l_2) & \to & k_1(h,l_1,l_2,append_1(l_1,l_2)) \\
k_1(h,l_1,l_2,out_1(l_3)) & \to & out_1(cons(h,l_3)) \\
\\
append_2(l) & \to & out_2(nil,l) \\
append_2(cons(h,l_3)) & \to & k_2(h,l_3,append_2(l_3)) \\
k_2(h,l_3,out_2(l_1,l_2)) & \to & out_2(cons(h,l_1),l_2) \\
\\
perm_0(nil) & \to & out_1(nil) \\
perm_i(l) & \to & k_{3,i}(l,append_2(l)) \\
k_{3,i+1}(l,out_2(v,cons(h,u))) & \to & k_{4,i+1}(l,h,v,u,append_1(v,u)) \\
k_{4,i+1}(l,h,v,u,out_1(w)) & \to & k_5(l,h,v,u,w,perm_i(w)) \\
k_5(l,h,v,u,w,out_1(t)) & \to & out_1(cons(h,t))
\end{array}$$

for all $i \in \mathbb{N}$. Although this TRS is infinite it can be proved terminating by RPO with precedence $perm_{i+1} \rhd k_{3,i+1} \rhd k_{4,i+1} \rhd perm_i$ for all $i \in \mathbb{N}$ and $perm_0 \rhd k_{3,0} \rhd k_5, append_1, append_2 \rhd k_1, k_2 \rhd out_1, out_2 \rhd cons, nil$.

4.5. EXAMPLE. The following logic program (see [Plü90, Example 5.1.2]) is an implementation of the well-known quick-sort algorithm.

$$\begin{array}{lcl}
more(s(x),0). & & \\
more(s(x),s(y)) & \leftarrow & more(x,y). \\
\\
lesseq(0,x). & & \\
lesseq(s(x),s(y)) & \leftarrow & lesseq(x,y). \\
\\
append(nil,l,l). & & \\
append(cons(h,l_1),l_2,cons(h,l_3)) & \leftarrow & append(l_1,l_2,l_3). \\
\\
split(h,nil,a,b,a,b). & & \\
split(h,cons(x,l),a,b,a_1,b_1) & \leftarrow & more(x,h), split(h,l,a,cons(x,b),a_1,a_2). \\
split(h,cons(x,l),a,b,a_1,b_1) & \leftarrow & lesseq(x,h), split(h,l,cons(x,a),b,a_1,a_2). \\
\\
qsort(nil,nil). & & \\
qsort(cons(h,l),s) & \leftarrow & split(h,l,nil,nil,a,b), qsort(a,a_1), qsort(b,b_1), \\
 & & append(a_1,cons(h,b_1),s).
\end{array}$$

The associated term rewriting system is given by:

$$\begin{array}{lcl}
more(s(x),0) & \to & out \\
more(s(x),s(y)) & \to & k_1(more(x,y)) \\
k_1(out) & \to & out \\
\\
lesseq(0,x) & \to & out \\
lesseq(s(x),s(y)) & \to & k_2(lesseq(x,y)) \\
k_2(out) & \to & out \\
\\
append(nil,l) & \to & out_1(l) \\
append(cons(h,l_1),l_2) & \to & k_3(h,append(l_1,l_2)) \\
k_3(h,out_1(l_3)) & \to & out_1(cons(h,l_3)) \\
\\
split(h,nil,a,b) & \to & out_2(a,b) \\
split(h,cons(x,l),a,b) & \to & k_4(h,x,l,a,b,more(x,h)) \\
k_4(h,x,l,a,b,out) & \to & k_5(split(h,l,a,cons(x,b))) \\
k_5(out_2(a_1,b_1)) & \to & out_2(a_1,b_1) \\
split(h,cons(x,l),a,b) & \to & k_6(h,x,l,a,b,lesseq(x,h)) \\
k_6(h,x,l,a,b,out) & \to & k_7(split(h,l,cons(x,a),b)) \\
k_7(out_2(a_1,b_1)) & \to & out_2(a_1,b_1) \\
\\
qsort(nil) & \to & out_1(nil) \\
qsort(cons(h,l)) & \to & k_8(h,split(h,l,nil,nil)) \\
k_8(h,out_2(a,b)) & \to & k_9(h,b,qsort(a)) \\
k_9(h,b,out_1(a_1)) & \to & k_{10}(h,a_1,qsort(b)) \\
k_{10}(h,a_1,out_1(b_1)) & \to & k_{11}(append(a_1,cons(h,b_1))) \\
k_{11}(out_1(s)) & \to & out_1(s)
\end{array}$$

We use the semantic labelling method to prove termination of this TRS. The model we construct is once again inspired by the length of lists, define the carrier set $M = \mathbb{N}$ and

$$\begin{array}{rcl@{\qquad}rcl}
nil_{\mathcal{M}} & = & 0 & k_{1,\mathcal{M}}(x) & = & 0 \\
cons_{\mathcal{M}}(h,t) & = & 1+t & k_{2,\mathcal{M}}(x) & = & 0 \\
0_{\mathcal{M}} & = & 0 & k_{3,\mathcal{M}}(x,y) & = & 1+y \\
s_{\mathcal{M}}(x) & = & x+1 & k_{4,\mathcal{M}}(h,x,l,a,b,z) & = & 1+l+a+b \\
out_{\mathcal{M}} & = & 0 & k_{5,\mathcal{M}}(x) & = & x \\
out_{1,\mathcal{M}}(x) & = & x & k_{6,\mathcal{M}}(h,x,l,a,b,z) & = & 1+l+a+b \\
out_{2,\mathcal{M}}(x,y) & = & x+y & k_{7,\mathcal{M}}(x) & = & x \\
more_{\mathcal{M}}(x,y) & = & 0 & k_{8,\mathcal{M}}(x,y) & = & 1+y \\
lesseq_{\mathcal{M}}(x,y) & = & 0 & k_{9,\mathcal{M}}(h,x,y) & = & 1+x+y \\
append_{\mathcal{M}}(x,y) & = & x+y & k_{10,\mathcal{M}}(h,x,y) & = & 1+x+y \\
split_{\mathcal{M}}(h,x,y,z) & = & x+y+z & k_{11,\mathcal{M}}(x) & = & x \\
qsort_{\mathcal{M}}(x) & = & x & & &
\end{array}$$

This defines a model, i.e., each left-hand side in the TRS has the same interpretation as the corresponding right-hand side. We choose label-sets S_{qsort}, S_{k_8} and S_{k_9} equal to the natural numbers and define $\pi_{qsort(x)} = x$, $\pi_{k_8(x,y)} = 1 + y$, $\pi_{k_9(h,x,y)} = 1 + x + y$ to label *qsort*. We choose label-sets S_{split}, S_{k_4} and S_{k_6} equal to the natural numbers and define $\pi_{split(h,l,a,b)} = l$, $\pi_{k_4(h,x,l,a,b,z)} = \pi_{k_6(h,x,l,a,b,z)} = l+1$ to label *split*. Now the rules with the predicates *split*, k_4 and k_6 are replaced by the infinitely many rules:

$$\begin{array}{lcl}
split_{\mathbf{0}}(h, nil, a, b) & \to & out_2(a, b) \\
split_{\mathbf{i+1}}(h, cons(x, l), a, b) & \to & k_{4,\mathbf{i+1}}(h, x, l, a, b, more(x, h)) \\
k_{4,\mathbf{i+1}}(h, x, l, a, b, out) & \to & k_5(split_{\mathbf{i}}(h, l, a, cons(x, b))) \\
split_{\mathbf{i+1}}(h, cons(x, l), a, b) & \to & k_{6,\mathbf{i+1}}(h, x, l, a, b, lesseq(x, h)) \\
k_{6,\mathbf{i+1}}(h, x, l, a, b, out) & \to & k_7(split_{\mathbf{i}}(h, l, cons(x, a), b))
\end{array}$$

for every $\mathbf{i} \in \mathbb{N}$. The rules with the predicates *qsort*, k_8 and k_9 of the TRS are replaced by the infinitely many rules:

$$\begin{array}{lcl}
qsort_{\mathbf{0}}(nil) & \to & out_1(nil) \\
qsort_{\mathbf{i}}(cons(h, l)) & \to & k_{8,\mathbf{i}}(h, split(h, l, nil, nil)) \\
k_{8,\mathbf{i}}(h, out_2(a, b)) & \to & k_{9,\mathbf{i}}(h, b, qsort_{\mathbf{j}}(a)) \\
k_{9,\mathbf{i}}(h, b, out_1(a_1)) & \to & k_{10}(h, a_1, qsort_{\mathbf{j}}(b))
\end{array}$$

for every $\mathbf{i}, \mathbf{j} \in \mathbb{N}$ with $0 \leq \mathbf{j} < \mathbf{i}$. To prove termination of this labelled TRS, use RPO with precedence: $qsort_{\mathbf{i+1}} \rhd k_{8,\mathbf{i+1}} \rhd k_{9,\mathbf{i+1}} \rhd qsort_{\mathbf{i}}$, $split_{\mathbf{i+1}} \rhd k_{4,\mathbf{i+1}} \rhd split_{\mathbf{i}}$ and $split_{\mathbf{i+1}} \rhd k_{6,\mathbf{i+1}} \rhd split_{\mathbf{i}}$ for every $\mathbf{i} \in \mathbb{N}$.

5. Using modularity results of constructor systems

Consider the following logic programs P_1:

$$\begin{array}{lcl}
add(0, x, x). & & \\
add(s(x), y, s(z)) & \leftarrow & add(x, y, z).
\end{array}$$

and P_2:

$$\begin{array}{lcl}
less(0, s(x)). & & \\
less(s(x), s(y)) & \leftarrow & less(x, y).
\end{array}$$

By modings $add(in, in, out)$ and $less(in, in)$ these programs are LR-well-moded. Intuitively one expects that the logic program given by both P_1 and P_2 terminates for all LR-well-moded queries, if and only if it the program P_1 terminates and the program P_2 terminates for all LR-well-moded queries. However, from the (innermost) termination of the TRS R_{P_1} and R_{P_2} we may not conclude (innermost) termination of the TRS $R_{P_1} \cup R_{P_2}$. Fortunately, the observation that with the algorithm of Section 2 the logic programs transform into constructor systems and some standard results for these constructor systems, give us the desired result.

5.1. THEOREM. *Let P be an LR-well-moded logic program and R_P the TRS obtained by applying the algorithm of Section 2 on P. Then R_P is a constructor system.*

PROOF. Let P be an LR-well-moded logic program. Define the set $\mathcal{D}$ as the set of all predicates p_i in P together with all function symbols k_i used in the translation. Define the set $\mathcal{O}ut$ as the set of all terms out_i used in the translation and $\mathcal{F}un$ as the set of all function symbols and constants of P. Let $\mathcal{C} = \mathcal{F}un \cup \mathcal{O}ut$.

It is easy to see from the algorithm that the left-hand side of a rewrite rule is of the form:

- $p_i(t_1, \ldots, t_n)$ for some i, or
- $k_i(Var, out_j(t_1, \ldots, t_n))$ for some i, j.

Since the terms $t_1, \ldots, t_n$ are terms of the logic program, they only contain variables, constants and function symbols of P. By definition these symbols are in $\mathcal{C}$. Since also out_j is an element of $\mathcal{C}$, the obtained TRS R_P is a constructor system. ∎

5.2. DEFINITION. If $(\mathcal{D}_1, \mathcal{C}_1, \mathcal{R}_1)$, $(\mathcal{D}_2, \mathcal{C}_2, \mathcal{R}_2)$ are constructor systems with $\mathcal{D}_1 \cap (\mathcal{C}_2 \cup \mathcal{D}_2) = \emptyset = \mathcal{D}_2 \cap (\mathcal{C}_1 \cup \mathcal{D}_1)$, then $(\mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{C}_1 \cup \mathcal{C}_2, \mathcal{R}_1 \cup \mathcal{R}_2)$ is said to be a *combination of constructor systems (with disjoint sets of defined symbols and common constructors).*

5.3. PROPOSITION ([Gra93]). *A combination of innermost terminating constructor systems with disjoint sets of defined symbols is again an innermost terminating constructor system.*

5.4. DEFINITION. Let P be a logic program. We say P is *partitioned in parts* $P_1, \ldots, P_n$ if

- each part P_i $1 \leq i \leq n$ is a subset of (program) clauses of P,
- every (program) clause of P occurs in at least one part P_i,
- The predicate symbols that occur in part P_i do not occur in the parts P_j for $j \neq i$. (Disjoint sets of predicate symbols).

5.5. THEOREM. *Let P be a LR-well-moded logic program and $P_1, \ldots, P_n$ the parts that are obtained by partitioning P. Let R_{P_i} be the associated rewrite system by part P_i for $1 \leq i \leq n$. If $R_{P_1}, \ldots, R_{P_n}$ are innermost terminating then the logic program P is terminating.*

Note that if P is an LR-well-moded logic program, then also all parts are LR-well-moded logic programs.

PROOF. Let P be a LR-well-moded logic program and $P_1, \ldots, P_n$ the parts that are obtained by partitioning P. By Theorem 5.1 all associated rewrite systems R_{P_i} with $1 \leq i \leq n$ are constructor systems. If $R_{P_1}, \ldots, R_{P_n}$ are innermost terminating, then by Proposition 5.3 R_P is innermost terminating. Hence by Corollary 3.4 P is terminating. ∎

Note that this modularity result does not hold for termination instead of innermost termination.

6. Structural recursive logic programs

In this section we prove that the TRS obtained by transforming a well-moded logic program is terminating for a wide class of logic programs: the structural recursive logic programs as defined by [Plü90].

6.1. DEFINITION. (cf. [Plü90, p. 9]) A p-clause is a clause of which the head has p as its predicate symbol. The subset of all p-clauses of a program is called the *procedure definition* π_p for p.

6.2. DEFINITION. (cf. [Plü90, p. 17]) For two predicates p and q defined in a program P, p is said to *depend* on q, written $p \to_\pi q$, if q occurs in the body of some of the clauses defining p. Let '$\to_\pi{}^+$' denote the transitive closure of '$\to_\pi$'. A predicate p is a *recursive predicate* if $p \to_\pi{}^+ p$ holds. Two predicates p and q with $p \neq q$ are said to be *mutually recursive* if $p \to_\pi{}^+ q$ and $q \to_\pi{}^+ p$.

6.3. DEFINITION. Let $>$ be an order on terms, we extend this order in a natural way to an order on sequences of terms (also denoted by $>$) in the following way:

$$\langle t_1, \ldots, t_n \rangle > \langle t'_1, \ldots, t'_n \rangle \quad \text{iff} \quad \begin{array}{ll} 1.\ t_j \geq t'_j \text{ for all} & j \in \{1, \ldots, n\} \\ 2.\ t_j > t'_j \text{ for some} & j \in \{1, \ldots, n\} \end{array}$$

Note that if the order on terms is well-founded, then the extended order on sequences of terms is also well-founded.

6.4. DEFINITION. (cf. [Plü90, p. 48] Let $\pi = \{C_1, \ldots, C_m\}$ be a recursive procedure definition for an n-ary predicate p in a well-moded logic program P which has no mutual recursion. Let $>$ be any well-founded order on terms, closed under substitution. The procedure π is said to be *structural recursive* if

there exists a set $I = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ of input indices of p such that
for all recursive clauses C_j with head $C_{j0} = p(t_1, \ldots, t_n)$ and
for all recursive literals C_{jr} in C_j with $C_{jr} = p(t'_1, \ldots, t'_n)$

we have that $\langle t_{i_1}, \ldots, t_{i_k} \rangle > \langle t'_{i_1}, \ldots, t'_{i_k} \rangle$. It is also said that the predicate p defined by π is structural recursive. We call the set of input indices I of a structural recursive predicate, the set of *decreasing arguments* of that predicate. A logic program is called structural recursive if all procedure definitions in the logic program are structural recursive.

6.5. EXAMPLE. The procedure definitions $append_1$ and $append_2$ in example 1.8 are structural recursive. For $append_1$ the set of decreasing arguments consists of the first argument, whereas it consists of the third argument for $append_2$. The procedure definition of *perm* in that example is not structural recursive. Since there is only one input index, viz. the first argument, there must be a well-founded order, closed under substitution, such that $l > w$. Such an order does not exist.

Note that $>$ is a well-founded order on all elements of the Herbrand universe HU_P, which is the same as $\mathcal{T}(\mathcal{F}un)$ (if necessary extended with a constant). In [Plü90, p. 49] is proved that all structural recursive logic programs terminate. In this section we prove that the transformation of a structural recursive logic program is terminating. We do not prove this directly for the obtained TRS R_P of a structural recursive logic program P, but for a labelled TRS $\overline{R}_P$ obtained from R_P by some construction called semantic labelling. To prove termination of R_P we use Proposition 1.20. To prove termination of the TRS $\overline{R}_P$ we use the following proposition, which is a direct consequence of well-foundedness of RPO.

6.6. PROPOSITION. *Let $\rhd$ be a well-founded order on the signature of a TRS R. If for every rule $l \to r$ in R we have that head$(l) \rhd f$ for all function symbols f that occur in r, then R is terminating.*

The following definitions are used to construct $\overline{R}_P$ from the TRS R_P corresponding to a structural recursive logic program P.

6.7. Definition. Let R_P correspond to a well-moded logic program P. We define a function *Chead* from k-symbols to terms inductively.

- if $p(t_1,\ldots,t_n) \to k_i(\ldots)$ is a rewrite rule of R_P, then $Chead(k_i) = p(t_1,\ldots,t_n)$,
- if $k_j(t_1,\ldots,t_n) \to k_i(\ldots)$ is a rewrite rule of R_P, then $Chead(k_i) = Chead(k_j)$.

By observation of the transformation algorithm one easily checks that by this definition *Chead* is well defined. We also define a function *Pred* from defined symbols to predicate symbols. For every predicate symbols p we define $Pred(p) = p$. For every k-symbols k_i we have $Chead(k_i) = p(\ldots)$ for some predicate symbol p and we define $Pred(k_i) = p$.

6.8. Example. For the defined symbols of the constructor system of Example 4.4 we have:

- $Chead(k_1) = append_1(cons(h, l_1), l_2)$,
- $Chead(k_2) = append_2(cons(h, l_3))$,
- $Chead(k_3) = Chead(k_4) = Chead(k_5) = perm(l)$.
- $Pred(k_1) = Pred(append_1) = append_1$,
- $Pred(k_2) = Pred(append_2) = append_2$,
- $Pred(k_3) = Pred(k_4) = Pred(k_5) = Pred(perm) = perm$.

6.9. Definition. Let R_P correspond to a well-moded logic program P. Now we define an algebra $\mathcal{M}_{R_P}$ (where the subscript R_P is omitted if it is clear which rewrite system we mean). Let the Herbrand universe HU_P be the carrier set of this model. We fix an arbitrary element c of HU_P.

- For every defined symbol d of arity n we define $d_{\mathcal{M}}(x_1,\ldots,x_n) = c$.
- For every symbol $out_n \in Out$ we define $out_{n\mathcal{M}}(x_1,\ldots,x_n) = c$.
- For every symbol $f \in \mathcal{F}un$ of arity n we define $f_{\mathcal{M}}(x_1,\ldots,x_n) = f(x_1,\ldots,x_n)$, where $x_1,\ldots,x_n$ quantify over HU_P.

with an interpretation function $[\![]\!] : \mathcal{T}(\mathcal{D} \cup Out \cup \mathcal{C}, \mathcal{V}) \times HU_P^{\mathcal{V}} \to HU_P$ from terms and valuations to the Herbrand universe:

$$\begin{array}{rcl} [\![x]\!]_\rho & = & \rho(x) \\ [\![f(t_1,\ldots,t_n)]\!]_\rho & = & f_{\mathcal{M}}([\![t_1]\!]_\rho,\ldots,[\![t_n]\!]_\rho) \end{array}$$

It is clear from this definition that the algebra $\mathcal{M}$ is a model for R_P, i.e. $[\![l]\!]_\rho = [\![r]\!]_\rho$ for all valuations ρ and all rules $l \to r$ in R_P.

In order to define $\overline{R}_P$, we first define the signature of $\overline{R}_P$. The constructor symbols of R_P are constructor symbols of $\overline{R}_P$ too. The set of defined symbols of R_P is extended to obtain the set of defined symbols of $\overline{R}_P$, denoted by $\overline{\mathcal{D}}$.

6.10. Definition. Let R_P correspond to a well-moded logic program P and $\mathcal{D}$ the set of defined symbols of R_P. We define the set $\overline{\mathcal{D}}$ as follows:

- If $d \in \mathcal{D}$ and $Pred(d)$ is a non-recursive predicate, then $d \in \overline{\mathcal{D}}$.
- If $d \in \mathcal{D}$ and $Pred(d)$ is a recursive predicate, then $d_{\mathbf{t}} \in \overline{\mathcal{D}}$ for every $\mathbf{t} \in HU_P$.

After the introduction of new symbols, the actual labelling of terms may take place.

6.11. DEFINITION. We define a labelling function $\mathsf{lab} : \mathcal{T}(\mathcal{D} \cup \mathcal{F}un \cup \mathcal{O}ut, \mathcal{V}) \times HU_P^{\mathcal{V}} \to \mathcal{T}(\overline{\mathcal{D}} \cup \mathcal{F}un \cup \mathcal{O}ut, \mathcal{V})$ by

- For every variable $x \in \mathcal{V}$ we define $\mathsf{lab}(x, \rho) = x$
- For every constructor symbol $f \in \mathcal{O}ut \cup \mathcal{F}un$ with arity n we define
$$\mathsf{lab}(f(t_1, \ldots, t_n), \rho) = f(\mathsf{lab}(t_1, \rho), \ldots, \mathsf{lab}(t_n, \rho))$$
- For every defined symbol $d \in \mathcal{D}$ with arity n and $Pred(d)$ is a non-recursive predicate, we define
$$\mathsf{lab}(d(t_1, \ldots, t_n), \rho) = d(\mathsf{lab}(t_1, \rho), \ldots, \mathsf{lab}(t_n, \rho))$$
- For every predicate symbol $d \in \mathcal{D}$ with arity n, $Pred(d)$ is a recursive predicate, $Chead(d) = p(s_1, \ldots, s_m)$, and $\{s_{i_1}, \ldots, s_{i_k}\}$ is the set of decreasing arguments of $p(s_1, \ldots, s_m)$, we define $l_\rho = \langle [\![s_{i_1}]\!]_\rho, \ldots, [\![s_{i_k}]\!]_\rho \rangle$ and
$$\mathsf{lab}(d(t_1, \ldots, t_n), \rho) = d_{l_\rho}(\mathsf{lab}(t_1, \rho), \ldots, \mathsf{lab}(t_n, \rho))$$

Let ε denote the empty sequence, then we also write d_ε for unlabelled defined symbols d.

6.12. DEFINITION. For the constructor system R_P over signature $\mathcal{D} \cup \mathcal{F}un \cup \mathcal{O}ut$ we define $\overline{R}_P$ to be the constructor system over signature $\overline{\mathcal{D}} \cup \mathcal{F}un \cup \mathcal{O}ut$ consisting of the rules
$$\mathsf{lab}(l, \rho) \to \mathsf{lab}(r, \rho)$$
for all valuations ρ and all rules $l \to r$ of R_P.

6.13. EXAMPLE. The following logic program P is obviously structural recursive
$$\begin{array}{lll} next(x, s(x)). & & \\ p(0). & & \\ p(s(x)) & \leftarrow & next(x, y), p(x). \end{array}$$
The corresponding TRS R_P is
$$\begin{array}{lll} next(x) & \to & out_1(s(x)) \\ p(0) & \to & out \\ p(s(x)) & \to & k_1(x, next(x)) \\ k_1(x, out_1(y)) & \to & k_2(x, p(x)) \\ k_2(x, out) & \to & out \end{array}$$
This TRS can not be proved terminating by RPO. Now we label the predicate p and all k-symbols that are related to p. The resulting (infinite) TRS is
$$\begin{array}{lll} next(x) & \to & out_1(s(x)) \\ p_{\mathbf{0}}(0) & \to & out \\ p_{\mathbf{s(t)}}(s(x)) & \to & k_{1,\mathbf{s(t)}}(x, next(x)) \\ k_{1,\mathbf{s(t)}}(x, out_1(y)) & \to & k_{2,\mathbf{s(t)}}(x, p_{\mathbf{t}}(x)) \\ k_{2,\mathbf{s(t)}}(x, out) & \to & out \end{array}$$
for every term $\mathbf{t}$ in the Herbrand universe of P. We can prove termination for this (infinite) TRS with RPO.

We prove that the above idea can be generalized, more precisely, $\overline{R}_P$ meets the requirements of Proposition 6.6. Therefore, we define an order on the signature, usually called a precedence and prove that this precedence is well-founded.

We introduce a relation on the defined symbols of a constructor system similar to the dependency relation $\rightarrow_\pi$ on the predicate symbols of a logic programs. This relation is the basis for the precedence that we construct.

6.14. DEFINITION. Let R be a constructor system. We define a relation $\rightsquigarrow$ on the set $\mathcal{D}$ of defined symbols as follows: If $l \rightarrow r$ is a rewrite rule of R and $d \in \mathcal{D}$ the defined symbol of the left-hand side l, then for all defined symbols $d' \in \mathcal{D}$ in the right-hand side r we define $d \rightsquigarrow d'$.

Let R_P correspond to a structural recursive logic program P, $\overline{R}_P$ defined as above. If $\rightsquigarrow$ as defined above is a relation on $\overline{R}_P$, then the following lemmas hold.

6.15. LEMMA. *If $k_{i,l} \rightsquigarrow k_{j,m}$, then $j = i + 1$ and $l = m$.*

PROOF. By the translation algorithm $j = i + 1$. Since $Chead(k_j) = Chead(k_i)$, $l = m$ by the definition of labelling. ∎

6.16. LEMMA. *If $p_l \rightsquigarrow k_{j,m}$, then $l = m$.*

PROOF. Since $Chead(k_j) = p(t_1, \ldots, t_n)$ for some terms $t_1, \ldots, t_n$ and l consists of a finite sequence of these terms, $l = m$ by the definition of labelling. ∎

6.17. COROLLARY. *If $p_l \rightsquigarrow^+ q_m$, then $p \rightarrow_\pi{}^+ q$.*

PROOF. Induction on the number of predicate symbols in-between and Lemma 6.15 and 6.16. ∎

6.18. LEMMA. *If $p_l \rightsquigarrow^+ p_m$, then $l > m$, where $>$ is the order on a finite sequence of terms.*

PROOF. Assume $p_l \rightsquigarrow^+ p_m$. Then there is a sequence of defined symbols $d_1, \ldots, d_n$ $(n \geq 0)$ such that

$$p_l \rightsquigarrow d_1 \rightsquigarrow \ldots \rightsquigarrow d_n \rightsquigarrow p_m$$

If one of the d_i $(1 \leq i \leq n)$ in the sequence is a (labelled) predicate symbol, q_k say, then by Corollary 6.17 $p \rightarrow_\pi{}^+ q$ and $q \rightarrow_\pi{}^+ p$. But the logic program is not mutual recursive, hence all defined symbols in the sequence are k-symbols. Thus by Lemma 6.15 and 6.16

$$p_l \rightsquigarrow k_{j,l} \rightsquigarrow k_{j+1,l} \rightsquigarrow \ldots \rightsquigarrow k_{j+n,l} \rightsquigarrow p_m$$

Hence by the transformation algorithm and the definition of the relation $\rightsquigarrow$, there are rewrite rules in $\overline{R}_P$ of the form

$$\begin{array}{lcl} p_l(t_1, \ldots, t_n) & \rightarrow & k_{j,l}(\ldots) \\ k_{j,l}(\ldots) & \rightarrow & k_{j+1,l}(\ldots) \\ & \vdots & \\ k_{j+n,l}(\ldots) & \rightarrow & k_{j+n+1,l}(\ldots, p_m(t'_1, \ldots, t'_n)) \end{array}$$

Since this is the translation of a structural recursive logic program, there is a index set $I = \{i_1, \ldots, i_k\}$, such that $l = \langle t_{i_1}, \ldots, t_{i_k} \rangle > \langle t'_{i_1}, \ldots, t'_{i_k} \rangle = m$. ∎

The precedence is defined by the relation $\rightsquigarrow$, extended from defined symbols to all symbols of the signature.

6.19. DEFINITION. Let $\overline{R}_P$, defined as above, correspond to a structural recursive logic program. We define a relation $\succ$ on the signature of $\overline{R}_P$ as follows:

- For every two elements $d_1, d_2 \in \overline{\mathcal{D}}$ we define $d_1 \succ d_2$ if $d_1 \rightsquigarrow d_2$.
- For every $d \in \overline{\mathcal{D}}$ and for all $out_i \in \mathcal{O}ut$ we define $d \succ out_i$.
- For every $out_i \in \mathcal{O}ut$ and for all $f \in \mathcal{F}un$ we define $out_i \succ f$.

Let $\rhd$ be the transitive closure of $\succ$.

6.20. LEMMA. *If P is a structural recursive logic program, R_P the TRS obtained from P and $\overline{R}_P$ as defined above, then the relation $\rhd$ as defined in Definition 6.19 is a well-founded partial order.*

PROOF. Assume there is an infinite sequence $f_1 \rhd f_2 \rhd \ldots$. Since there are only finitely many constructor symbols, which are all smaller than the defined symbols, the sequence consists of infinitely many defined symbols $d_1 \rhd d_2 \rhd \ldots$. Thus by definition, there is an infinite sequence $d_1 \rightsquigarrow d_2 \rightsquigarrow \ldots$ of defined symbols. By Lemma 6.15 and the fact that there are only finitely many k-symbols in the unlabelled set of defined symbols, there is no infinite sequence of only k-symbols. Thus, every infinite sequence contains infinitely many labelled predicate symbols. Since there are only finitely many predicate symbols in the unlabelled set of defined symbols, there is one predicate symbol that occurs infinitely many times in this sequence. Thus, $p_{l_1} \rightsquigarrow^+ p_{l_2} \rightsquigarrow^+ p_{l_3} \rightsquigarrow^+ \ldots$ and by Lemma 6.18 $l_1 > l_2 > l_3 > \ldots$. This contradicts that the term order $>$ is well-founded. Thus, $\rhd$ is well-founded. ∎

Since the precedence is well-founded we can now prove our main result.

6.21. THEOREM. *If R_P is a TRS corresponding to a structural recursive logic program P, then R_P terminates.*

PROOF. By Proposition 1.20 it suffices to prove that $\overline{R}_P$ terminates. By definition of the relation $\rhd$ for every rule $l \to r$ in $\overline{R}_P$ we have that $head(l) \rhd f$ for all function symbols f that occur in r. Hence, by Proposition 6.6 $\overline{R}_P$ terminates. ∎

The property we used from Definition 6.3 is that the order on sequences preserves well-foundedness. Hence, the definition of structural recursive logic programs is easily generalized by taking an arbitrary well-foundedness-preserving order on sequences of terms, like the lexicographic order or the multiset order. Without any extra effort Theorem 6.21 holds for this extension. As an example we take Ackermann's function written as a logic program

$$\begin{array}{lll} ack(0,x,s(x)). & & \\ ack(s(x),0,y) & \leftarrow & ack(x,s(0),y). \\ ack(s(x),s(y),z) & \leftarrow & ack(s(x),y,w), ack(x,w,z). \end{array}$$

If we choose the lexicographic order the requirements for (extended) structural recursive programs are easily verified. Hence by Theorem 6.21 the TRS corresponding to this program terminates. Hence the logic program terminates.

7. Conclusions

We presented a simply implementable transformation from logic programs into TRSs such that termination of the logic program follows from (innermost) termination of the TRS. Analyzing termination of a TRS is much more basic than termination of a logic program since it does not depend on a particular computation rule and no unification is involved. As a technique for automatically proving termination of logic programs this is very promising. We proved that all TRSs obtained from an extension of the class of structural recursive logic programs are indeed terminating. For other kind of programs we gave some examples. It turns out that the basic techniques of proving termination of TRSs, like RPO, only cover a small class of logic programs. However if some labelling is applied on the TRS obtained from a logic program, motivated on semantic information of ingredients of the logic program like lengths of lists, then often a TRS is obtained for which RPO succeeds in proving termination. Extracting the right semantic information and choosing the right labelling is subject of further research, of which the ultimate goal consists of an implementation having logic programs as input and termination proofs as output.

References

[AM93] G. Aguzzi and U. Modigliani. Proving termination of logic programs by transforming them into equivalent term rewriting systems. *Proceedings of FST&TCS 13*, Lecture Notes in Computer Science(761), 12 1993.

[CR] Maher Chtourou and Michaël Rusinowitch. Méthode transformationnelle pour la preuve de terminaison des programmes logiques. unpublished manuscript in French.

[DW88] Saumya K. Debray and David S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5:207–229, 1988.

[FZ94] Maria Ferreira and Hans Zantema. Syntactical analysis of total termination. *Proceedings of ALP'94*, Lecture Notes in Computer Science, 1994. To appear.

[Gra93] Bernhard Gramlich. Relating innermost, weak, uniform and modular termination of term rewriting systems. Technical Report SR-93-09, Universität Kaiserslautern, June 1993.

[GW92] Harald Ganzinger and Uwe Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. *Proceedings of CTRS*, Lecture Notes in Computer Science(656):430–437, July 1992.

[KKS91] M.R.K. Krishna Rao, Deepak Kapur, and R.K. Shyamasundar. A transformational methodology for proving termination of logic programs. *Proceedings of CSL*, Lecture Notes in Computer Science(626):213–226, 1991.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second extended edition, 1987.

[MT91] Aart Middeldorp and Yoshihito Toyama. Completeness of combinations of constructor systems. *Proceedings of RTA-91*, Lecture Notes in Computer Science(488):188–199, April 1991.

[Plü90] Lutz Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1990. Subseries of Lecture Notes in Computer Science.

[Ste92] Joachim Steinbach. Notes on transformation orderings. Technical Report SR-92-23, Universität Kaiserslautern, 1992.

[Zan93a] H. Zantema. Termination of term rewriting by interpretation. *Proceedings third international workshop CTRS-92*, Lecture Notes in Computer Science(656):155–167, 1993. Full version appeared as report RUU-CS-92-14, Utrecht University.

[Zan93b] Hans Zantema. Termination of term rewriting by semantic labelling. Technical Report RUU-CS-93-24, Utrecht University, July 1993. Accepted for special issue on term rewriting of Fundamenta Informaticae.