Proving Existential Termination of Normal Logic Programs

Massimo Marchiori
Dept. of Pure and Applied Mathematics, University of Padova
Via Belzoni 7, 35131 Padova, Italy
max@hilbert.math.unipd.it

Abstract. The most important *open problem* in the study of termination for logic programs is that of *existential termination*. In this paper we present a powerful transformational methodology that provides necessary (and, under some conditions, sufficient) criteria for existential termination. The followed approach is to develop a suitable transformation from logic programs to Term Rewriting Systems (TRSs), such that proving termination of the obtained TRS implies existential termination of the original logic program. Thus, all the extensive amount of work on termination for TRSs can be automatically used in the logic programming setting. Moreover, the approach is also able to cope with the dual notion of universal termination: in fact, a whole spectrum of termination properties, said k-termination, is investigated, of which universal and existential termination are the extremes. Also, a satisfactory treatment to the problem of termination for logic programming with *negation* is achieved. This way we provide a unique, uniform approach covering all these different notions of termination.

1 Introduction

The study of program termination is a fundamental topic in computer science. In the field of logic programming, however, the power of the paradigm, together with the way in which it is implemented (e.g. in Prolog), make the study of termination extremely hard. Two kinds of termination are distinguished for logic programs: existential and universal.

The key property of *existential termination* is the natural notion of termination from the programmer's viewpoint: if the program is run with an input, it must stop (finding a solution to the problem or saying there are not solutions).

Unfortunately, existential termination is still the most important *open problem* (see [13]) in the field of termination for logic programs. Very few works so far tried to shed some light on the problem, namely [9, 18, 6], without giving satisfactory results (cf. [13]): all of them give results of expressibility nature, saying that the Prolog operational semantics can be in principle codified into some formalism, like first-order logic for instance, and thus termination and other properties could be studied by trying to use some kind of inductive reasoning or the like.

On the other hand, the 'dual' notion of *universal termination* is the much stronger property that says a program must terminate not only existentially, but *for every* further invocation, by the user, of backtracking, and moreover that the number of solutions to the problem must be finite.

This property has been the subject of a great number of works (cf. [13]) but, due to the intrinsic complexity of the problem, even in this much more restrictive case, most of the works are only of theoretical nature and extremely difficult to implement.

A noticeable exception is given by the so-called 'transformational approach' started by Rao, Kapur and Shyamasundar in [26] and further investigated in [19, 1, 8, 23, 5], consisting in giving a transformation from logic programs into TRSs such that to prove the universal termination of a logic program it suffices to prove the termination of the transformed term rewriting system.

This transformational approach has several advantages. The main one is that for TRSs the study of termination, in sharp contrast to the logic programming case, is much easier, being available plenty of powerful criteria and many automatic or semi-automatic implementations to test termination: for instance path orderings, polynomial orderings, semantic labelling, general path orderings and many others (see e.g. [15, 17, 16]). The reader is referred to [27] for a nice application of the transformational approach to compiler verification.

Another advantage of this approach is that giving such a translation we do not obtain only *one* criterion but a bunch of: every (present or future) criterion of termination for TRSs becomes automatically a criterion for logic programs

In this paper, we address the open problem of existential termination by developing a suitable powerful transformational approach able to cope with this fundamental property. This way, we also gain all the aforementioned benefits proper of this kind of approach.

In fact, we will tackle a much more general problem, introducing and studying the more expressive property of *k-termination*: roughly speaking, given an ordinal k a program k-terminates if its first k derivations are finite. k-termination generalizes both existential and universal termination (corresponding respectively to 1-termination and $\omega + 1$-termination), providing a hierarchy of intermediate properties.

We also show how the presented method can cope without difficulties even with the corresponding *strong* versions of termination (cf. [13]), i.e. termination not only w.r.t. one input but w.r.t. all the possible inputs.

This way, we provide a unique, *uniform* way to cope with all these different notions of termination.

Moreover, we do not limit to definite logic programming, but we cover also termination of *normal logic programming*, i.e. programs with the important feature of *negation*, as implemented in Prolog. The primary importance of negation for applications in non-monotonic reasoning and in artificial intelligence is well-known. However, even in the restricted ambit of universal termination a fully satisfactory treatment of termination of programs with negation has been so far out of scope, since the problem is tightly related to existential termination: for instance, a program universally terminates w.r.t. a ground literal $not\, A$ if and only if it existentially terminates w.r.t. A.

The analysis is taken even further: it is carefully studied to what extent we get not only *sufficient* criteria for all these kinds of termination, but even *necessary* ones, thus allowing to formally state what is the 'minimum power' of the method.

So, for instance, the presented method, when restricted to universal termination only, is *by far* more powerful than all the other works based on the transformational approach.

Another point is that, unlike the other works based on the transformational approach, here we followed a modular technique: Instead of presenting a very complicated transformation for a main class of logic programs, we built the transformation as a composition of two smaller submodules. This way we split the complexity of a big transformation into a composition of two easier sub-transformations, making the analysis easier; also, subsequent improvements can be obtained separately enhancing one

of the submodules, without having to rebuild a whole transformation from the scratch.

The work is organized as follows. First, we develop a transformation to TRSs for a core subclass of logic programs, that of *Regularly Typed* programs (RT for short). This core transformation is proven to completely preserve k-termination, hence giving a *necessary and sufficient* criterion for k-termination (or, better, plenty of them, for what we said earlier). We then show how this subclass can be extended to the bigger class of *Safely Typed* programs (ST), via a suitable transformation (which is of independent interest) from ST logic programs to RT logic programs. Then, all the results are extended to normal logic programming, thus covering *negation*. Finally, an accurate comparison with the related work is presented.

2 Preliminaries

We assume basic knowledge of logic programming and term rewriting systems. For standard logic programming terminology, we will mainly follow [21], whilst for TRSs we use standard notations from [17]. Logic programs will be considered as executed with leftmost selection rule and depth-first search rule, that is the standard way in which logic programming is implemented (for example, in Prolog). Also, we will consider in full generality conditions that can constrain both the logic program and the goal: so, for notational convenience we will talk by abuse of a *class of logic programs* meaning a collection both of logic programs and of goals.

2.1 Notation

We assume that the logic program is written using the (infinite) set of variables VAR and a signature $\Sigma = \{p_0, p_1, .. ; f_0, f_1, ..\}$ where p_i are the predicate symbols and f_i the function symbols (constants are nullary functions). Usually, the employed Σ will be just the minimal signature in which the considered logic program can be written, hence a finite one.

Given a substitution ϑ, $Dom(\vartheta)$ and $Ran(\vartheta)$ indicates, respectively, its domain and range; ϑ^{-1} denotes its inverse mapping, and $\vartheta|_V$ its restriction to some set of variables V. Composition of two functions f and g will be indicated with $f \circ g$. Sequences of terms will be written in vectorial notation (e.g. $\overline{t}$). Sequences in formulae should be seen just as abbreviations: for instance, $[\overline{t}]$, with $\overline{t} = t_1, .., t_m$, denotes the string $[t_1, .., t_m]$. Accordingly, given two sequences $\overline{s} = s_1, .., s_n$ and $\overline{t} = t_1, .., t_m$, $\overline{s}, \overline{t}$ stands for the sequence $s_1, .., s_n, t_1, .., t_m$.

Given a family S of objects (terms, atoms, etc.), $\mathrm{Var}(S)$ is the set of all the variables contained in it; moreover, S is said to be *linear* if no variable occurs more than once in it. For every term (or sequence) t, a *linearization* of t (via σ) is a linear term (sequence) t' such that, for some substitution σ, $t'\sigma = t$, $Dom(\sigma) = \mathrm{Var}(t')$, $\mathrm{Var}(t') \cap \mathrm{Var}(t) = \emptyset$, and $Ran(\sigma) \subseteq$ VAR (i.e., we simply replace repeated variables with different fresh ones to make the term linear: for instance, if $t = f(X, g(X, Y))$ we could take $t' = f(Z, g(V, W))$ and $\sigma = \{Z/X, V/X, W/Y\}$).

To make formulae more readable, we will sometimes omit brackets from the argument of unary functions (e.g. $f(g(X))$ may be written fgX). Also, given a sequence $\overline{t} = t_0, .., t_n$ and a unary function f, we use $f\overline{t}$ as a shorthand for $f(t_0), .., f(t_n)$.

2.2 Goals as Clauses

Being goals and clauses different objects, when describing a class of logic programs one usually has to provide different descriptions both for the goal and for the clauses. In this paper we will overcome this difficulty using the following definition:

Definition 2.1 A class $\mathcal{P}$ is said *regular* if $P \cup \{\leftarrow A_1, .., A_m\} \in \mathcal{P} \Leftrightarrow P \cup \{goal \leftarrow A_1, .., A_m\} \in \mathcal{P}$ (where *goal* is a new nullary predicate symbol). □

Using regular properties allows to define a class of logic programs and goal giving only the definition for programs, hence making definitions much shorter.

Assumption 1 All the classes we consider in this paper are understood to be regular.

In the context of this paper, this will be even more useful: since we are going to introduce transformations that translate logic programs (possibly together with a goal) into logic programs (possibly with a goal) or into TRSs (possibly together with a term), we can again shorten the definitions of such transformations by defining them only on logic programs: goals G are identified with the clause $goal \leftarrow G$ (and analogously, for TRSs terms t are identified with a produced 'rule' of the form $goal \rightarrow t$). This automatically gives a translation for the goal(s) eventually present.

3 The Program Classes

Definition 3.1 A *mode* for a n-ary predicate p is a map from $\{1, .., n\}$ to $\{in, out\}$. A *moding* is a map associating to every predicate p a mode for it. A *moded program* is a program endowed with a moding. An argument position of a moded predicate is called input (resp. output) if it is mapped by the mode into *in* (resp. *out*). □

Multiple modings can be defined by renaming the predicates.

$p(\bar{s}; \bar{t})$ denotes a moded atom p having its input positions filled in by the sequence of terms $\bar{s}$, and its output positions filled in by $\bar{t}$. We denote with $\mathrm{in}(p)$ and $\mathrm{out}(p)$ respectively the number of input and output positions of p.

A moded predicate should be roughly seen as a function from its input arguments to its output ones. For instance, a predicate p with moding (in, in, out) should be viewed as a function having two inputs (the first two arguments) and one output (the third one).

The programs that we will consider are typed. Any type system can be used, provided only it satisfies the following:

Assumption 2 Every type is closed under substitutions.

We denote with *Types* the set of types used in the chosen type system. For example, possible types are Any (all the terms), Nat (the terms 0, $s(0)$, $s(s(0))$, ..), $Ground$ (all the ground terms), $List$ (all the lists), $NatList$ (all the lists of Naturals) and so on. In the following examples we will assume these basic types are in the type system. Also, we say a type is ground if it is contained in $Ground$.

A term t of type T will be indicated with $t : T$. If $\bar{t} = t_1, .., t_n$ and $\bar{T} = T_1, .., T_n$ are respectively a sequence of terms and types, $\bar{t} : \bar{T}$ is a shorthand for $t_1 : T_1, .., t_n : T_n$.

Just like modes, types can be associated to predicates as well:

Definition 3.2 A *type* for an n-ary predicate p is a map from $\{1, .., n\}$ to *Types*. A *typing* is a map associating to every predicate p a type for it. A *typed program* is a program endowed with a typing. An argument position of a typed predicate is said of type T if it is mapped by the type into T. □

We write $p(m_1 : T_1, .., m_n : T_n)$ to indicate that a predicate p has moding $(m_1, .., m_n)$ and typing $(T_1, .., T_n)$.

To reason about types, we employ the standard concept of *type checking*: an expression of the form $\bar{s} : \bar{S} \models \bar{t} : \bar{T}$ indicates that from the fact that $\bar{s}$ has type $\bar{S}$ we can infer that $\bar{t}$ has type $\bar{T}$. More formally, $\bar{s} : \bar{S} \models \bar{t} : \bar{T}$ if for every substitution θ, $\bar{s}\theta \in \bar{S}$ implies $\bar{t}\theta \in \bar{T}$. For instance, $X : Any, Y : List \models [X|Y] : List$.

Another concept we need is the following:

Definition 3.3 A term t is a *generic expression* of the type T if every $s \in T$ having no common variables with t and unifying with it is an instance of t (i.e. $\exists\theta.\, t\theta = s$). □

For example, variables are generic expressions of *Any*, every term is a generic expression of *Ground*, $[], [X], [X|X], [X|Y], [X, Y|Z]$ etc. are generic expressions of *List*.

We will use types and generic expression in such a way that during a program execution unification behaves in a more regular way, that is to say it can be performed using repeated applications of pattern matching (see [3, 24]). So, we now introduce the main class studied in the paper:

Definition 3.4 A program is said to be *Safely Typed* (ST) if for each of its clauses $p_0(\bar{t}_0 : \bar{T}_0; \bar{s}_{n+1} : \bar{S}_{n+1}) \leftarrow p_1(\bar{s}_1 : \bar{S}_1; \bar{t}_1 : \bar{T}_1), .., p_n(\bar{s}_n : \bar{S}_n; \bar{t}_n : \bar{T}_n)$ we have:
• $\bar{t}_0 : \bar{T}_0, .., \bar{t}_{j-1} : \bar{T}_{j-1} \models \bar{s}_j : \bar{S}_j \quad (j \in [1, n+1])$
• each term in $\bar{t}_i$ is filled in with a generic expression for its corresponding type in $\bar{T}_i$
• if a variable X occurs twice in $\bar{t}_0, .., \bar{t}_n$, then there is a $\bar{t}_i$ $(0 \leq i \leq n)$ s.t. $X \in \mathrm{Var}(\bar{t}_i)$, $X \notin \mathrm{Var}(\bar{t}_0, .., \bar{t}_{i-1})$, and every term $r \in \bar{t}_i$ has a corresponding ground type. □

For example, the program quicksort using difference lists (see Example 6.2) is Safely Typed. The scope of the class ST is quite large: it is comparable to the class of *Well Typed* programs introduced in [7]; for instance, the great majority of the programs in [29] and [10] are safely typed. Finding whether a program is ST or not is a problem that can be addressed using one of the many existing tools to find moding and typing information of a logic program (e.g. [14, 28, 2]). Moreover, the syntactical nature of the class makes it suitable to be used just as a strongly typed logic programming language on its own. This is the direction followed in many recent systems: in many cases the moding/typing information can be optionally supplied; in others, like the state-of-the-art fastest compiler, Mercury (cf. [12]), modes and types are just the adopted syntax.

We note how, when the type system contains only the type *Ground*, the ST class collapses into the well-known class of *Well Moded* programs (cf. [7]).

Definition 3.5 A program is said to be *Regularly Typed* (RT) if it is Safely Typed and for each of its clauses $p_0(\bar{s}_0; \bar{t}_0) \leftarrow p_1(\bar{s}_1; \bar{t}_1), .., p_n(\bar{s}_n; \bar{t}_n)$ we have that $\bar{t}_1, .., \bar{t}_n$ is a linear sequence of variables and $\forall i \in [1, n]$. $\mathrm{Var}(\bar{t}_i) \cap \bigcup_{j=0}^{i} \mathrm{Var}(\bar{s}_j) = \emptyset$. □

Example 3.6 The usual program to add two numbers

$$add(0, X, X) \leftarrow \qquad\qquad add(s(X), Y, s(Z)) \leftarrow add(X, Y, Z)$$

with moding/typing $add(in : Ground, in : Any, out : Any)$ is regularly typed.

Also, the standard basic programs *append*, *reverse*, *quicksort*, *member* etc. are (with suitable modings/typings) all in RT. □

It is interesting to notice that *many* parts of logic programming codes are written, more or less consciously, in the form given by the RT class. Indeed, this class properly contains the class of simply moded and well typed (SWT) programs introduced in [3], and that class has already been shown to be quite expressive (see for instance the list of programs presented in [3]).

We remark how the above definitions concern *definite* logic programs only (i.e. programs without negation). In Section 8 these classes will be extended to *normal* logic programs (i.e. programs with negation).

4 k-termination

Suppose a logic program P is run with goal G. Let us denote with $answer_{P,G}(1)$ the first obtained answer: it is equal to

1. ϕ if the computation terminates successfully giving ϕ as computed answer substitution.
2. Fail if the computation terminates with failure.
3. $\perp$ if the computation does not terminate.
(here, Fail and $\perp$ are special symbols used to denote failure and nontermination respectively).

In case 1, the user can activate backtracking to look for the second answer $answer_{P,G}(2)$, and so on till for some $k \geq 1$ $answer_{P,G}(k)$ returns Fail or $\perp$ (in case of infinite answers, we assume $k = \omega$ and $answer_{P,G}(\omega) = \perp$).

Now, the *answer semantics* $\boldsymbol{\alpha}_P(G)$ of a logic program P w.r.t. a goal G is defined as the (possibly infinite) sequence

$$\boldsymbol{\alpha}_P(G) = \ answer_{P,G}(1), .., answer_{P,G}(k)$$

We can now provide a formal definition of termination:

Definition 4.1 Given a program P and a goal G, suppose its answer semantics is $\boldsymbol{\alpha}_P(G) = \phi_0, .., \phi_m$. Then P is said to *existentially (resp. universally) terminate* w.r.t. G if $\phi_0 \neq \perp$ (resp. if $\phi_m \neq \perp$). □

Hence, a program existentially terminates if its first answer is different from $\perp$ (i.e. it is not an infinite derivation), and universally terminates if it does not give $\perp$ answers at all (i.e. the program returns a finite number of answers and then halts with a failure).

There is however a more general concept of termination, that encompasses the previous two:

Definition 4.2 Given a program P and a goal G, suppose its answer semantics is $\boldsymbol{\alpha}_P(G) = \phi_0, .., \phi_m$. Then, for every ordinal k, P is said to *k-terminate* w.r.t. G if $\forall i < k.\ \phi_i \neq \perp$. □

k-termination provides a complete spectrum of termination properties, with intermediate degrees between the two extremes consisting of existential and universal termination. Indeed, it is immediate to see that existential termination corresponds to 1-termination, whereas universal termination corresponds to $\omega + 1$-termination. Note that for every ordinal $k > \omega + 1$, k-termination coincides with $\omega + 1$-termination, hence universal termination is the strongest termination property in this hierarchy. Observe also that every program trivially 0-terminates, and hence we can without loss of generality restrict our attention to k-termination with $1 \leq k \leq \omega + 1$.

Example 4.3 The termination property closest to universal termination in the k-termination hierarchy is ω-termination, that says a program cannot enter an infinite derivation (but might perform an infinite number of finite derivations). □

4.1 Strong k-termination

In this paper we will also investigate strong k-termination, that is k-termination not only for a single goal, but for all the goals in the given class:

Definition 4.4 Given a class $\mathcal{P}$ of logic programs, and an ordinal k, a program $P \in \mathcal{P}$ is said to *strongly k-terminate* w.r.t. $\mathcal{P}$ if P k-terminates w.r.t. G for every goal $G \in \mathcal{P}$. □

The big difference with the previous case of k-termination w.r.t. a goal is given by this result (to be precise, we remark that it holds under the assumption of *persistent classes* (i.e. closed via resolution, see [24]), an assumption always satisfied in this paper):

Theorem 4.5 *Strong existential termination and strong ω-termination coincide.*

That is to say, in the strong termination case the k-termination hierarchy *collapses* into two properties only (plus the trivial strong 0-termination): strong existential and strong universal termination.

In the sequel, when talking about strong termination w.r.t. some class $\mathcal{P}$, we will usually omit mentioning $\mathcal{P}$: it will be clear from the context what class is meant.

5 The Basic Transformation

In this section we provide the transformation $\mathcal{E}_{\mathrm{RT}}$ from regularly typed program to TRSs that will be the core of the subsequent transformations. Before giving the formal definition, we need some preliminary notions.

In the corresponding TRS we will utilize, besides the symbols of the original logic program, some new symbols.

We will employ so-called $\Diamond$-*lists*, that is lists where the constructors are the binary symbol c and the constant $\Diamond$: we will use the notation $\langle t_1, .., t_n \rangle$ to denote such lists (e.g. $\langle t_1, t_2 \rangle = \mathsf{c}(t_1, \mathsf{c}(t_2, \Diamond))$).

The unary symbol $\mathcal{M}$ will be used as a marker to indicate that its argument is, roughly speaking, a 'result' (i.e. a datum that doesn't need to be processed further). Also, we will make use of symbols of the form $\lambda_{t_1}^{t_2}$, that can be roughly seen as the function $\lambda t_1.t_2$ (i.e. it expects a datum of the form t_1 and gives as output t_2): the exact formalization of this 'lambda operator' will be given later.

Definition 5.1 Take a regularly typed clause $C = p_0(\bar{t}_0; \bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1; \bar{t}_1), .., p_n(\bar{s}_n; \bar{t}_n)$. Then $\mathsf{FLOW}(C)$ is defined as

$$\lambda^{[\mathcal{M}\bar{s}_{n+1}]}_{[\mathcal{M}\,V(n),\mathcal{M}\bar{t}_n]} \lambda^{[\mathcal{M}\,V(n)\,|\,p_n[\mathcal{M}\bar{s}_n]]}_{[\mathcal{M}\,V(n-1),\mathcal{M}\bar{t}_{n-1}]} \cdots \lambda^{[\mathcal{M}\,V(1)\,|\,p_1[\mathcal{M}\bar{s}_1]]}_{[\mathcal{M}\,V(0),\mathcal{M}\bar{t}_0]}[\mathcal{M}\bar{t}_0]$$

where $V(k) = \cup_{i=0}^{k-1} \mathrm{Var}(\bar{t}_i)$. □

The idea behind the FLOW definition is that every clause $p_0(\bar{t}_0; \bar{s}_{n+1}) \leftarrow ..$ provides a way to calculate $p_0[\mathcal{M}\bar{t}_0]$ (i.e. p_0 applied to its input arguments). Its output value $[\mathcal{M}\bar{s}_{n+1}]$ is obtained in the following way.

Informally, $V(k)$ denotes the Variables of $p_1, .., p_{k-1}$ that could be needed for the input arguments of $p_{k+1}, .., p_n$ and for the output argument of the head predicate p_0 (i.e. $\bar{s}_{k+1}, .., \bar{s}_{n+1}$).

We start with the input data $[\mathcal{M}\bar{t}_0]$. Then, applying the first operator $\lambda^{[\mathcal{M}\,V(1)\,|\,p_1[\mathcal{M}\bar{s}_1]]}_{[\mathcal{M}\,V(0),\mathcal{M}\bar{t}_0]}$ we calculate $p_1[\mathcal{M}\bar{s}_1]$ (that gives its output values for $\mathcal{M}\bar{t}_1$), together with the values from $\mathcal{M}\bar{t}_0$ that are needed in the sequel to calculate some other $p_i[\mathcal{M}\bar{s}_i]$ or the final output $[\mathcal{M}\bar{s}_{n+1}]$ (i.e. $V(1)$). The process goes on till all the $p_1, .., p_n$ have been processed, and the last operator $\lambda^{[\mathcal{M}\bar{s}_{n+1}]}_{[\mathcal{M}\,V(n),\mathcal{M}\bar{t}_n]}$ simply passes to the final output $[\mathcal{M}\bar{s}_{n+1}]$ the values previously computed (present in $[\mathcal{M}\,V(n), \mathcal{M}\bar{t}_n]$).

Example 5.2 After Example 3.6, let C be the clause $add(s(X), Y, s(Z)) \leftarrow add(X, Y, Z)$ (recall the moding/typing was $add(in\colon Ground, in\colon Any, out\colon Any)$). Then

$$\mathsf{FLOW}(C) \;=\; \lambda^{[\mathcal{M}s(Z)]}_{[\mathcal{M}X,\mathcal{M}Y,\mathcal{M}Z]} \lambda^{[\mathcal{M}X,\mathcal{M}Y\,|\,add[\mathcal{M}X,\mathcal{M}Y]]}_{[\mathcal{M}s(X),\mathcal{M}Y]}[\mathcal{M}s(X), \mathcal{M}Y] \qquad \square$$

Definition 5.3 The map $\mathcal{V}$ from terms to terms is inductively defined this way:

$$\begin{array}{ll} \mathcal{V}(f(t_1, .., t_k)) = f(\mathcal{V}(t_1), .., \mathcal{V}(t_k)) & (f \in \Sigma) \\ \mathcal{V}(X) = \mathsf{v} & (X \in \textsc{Var}) \end{array}$$

where v is a special new constant. □

Hence, the map $\mathcal{V}$ simply replaces every variable of a term with the special constant v: for instance, $\mathcal{V}(f(X, g(Y, a))) = f(\mathsf{v}, g(\mathsf{v}, a))$.

Definition 5.4 (Unification Engine)
For every term t, its unification engine $UNIFY_t$ is defined as follows. Let t' be a linearization of t (via σ). Then the rules defining $UNIFY_t$ are:

$$UNIFY_t(X) \to \mathcal{U}_t(X, \mathcal{L}(\mathcal{V}(t), X))$$
$$\mathcal{U}_t(t', true) \to \bigwedge_{\{X_1, \ldots, X_k\} = \sigma^{-1}(X)} \bigwedge_{X \in \mathrm{Var}(t)} \bigwedge_{i=1 \ldots k-1} \mathcal{L}(X_i, X_{i+1})$$
$$\mathcal{U}_t(X, false) \to false$$
$$\mathcal{L}(\mathsf{v}, X) \to true$$
$$\mathcal{L}(f(X_1, .., X_m), f(Y_1, .., Y_m)) \to \wedge_{i=1..m} \mathcal{L}(X_i, Y_i) \qquad (f \in \Sigma)$$
$$\mathcal{L}(f(X_1, .., X_m), g(Y_1, .., Y_n)) \to false \qquad (f, g \in \Sigma,\ f \not\equiv g)$$
$$true \wedge true \to true$$
$$X \wedge false \to false$$
$$false \wedge X \to false$$

(note that we write $\wedge_{i \in \emptyset}$ as a synonymous for $true$). □

The unification engine of a term t formalizes in the TRS the concept of unification: it tests whether or not a given term is unifiable with t.

Informally, the behaviour of $UNIFY_t$ can be summarized as follows. The $\mathcal{L}$ test performs a kind of restricted Martelli-Montanari algorithm, as can be easily seen looking at the rewrite rules defining it: roughly speaking, it performs unification of $\mathcal{L}$inear terms. The only rule not immediate to understand is $\mathcal{L}(\mathsf{v}, X) \to true$: it simply says that whenever the first argument is a variable (denoted by the special constant v), then everything unifies with it. This is the reason why the $\mathcal{V}$ operator was introduced (Definition 5.3): it performs the 'is-a-variable' test at a syntactic level.

$UNIFY_t$ invokes $\mathcal{L}$ several times, since it must also face the problem of all the repeated variables (i.e. non-linear terms): this is done in the rules defining $\mathcal{U}_t$, where repeated variables are in sequence, via an 'and' operator (written infix for easier readability) imposed to have a common unifier.

Note that the unification engine is built to work with the terms produced by the transformation only (i.e. when invoked in the transformation it properly performs the unification test, but it does not work in general for all the terms).

Example 5.5 Take the term $t = f(X, g(X, Y))$, and a corresponding linearization $t' = f(Z, g(V, W))$. Then the first two rules defining $UNIFY_{f(X,g(X,Y))}$ are:

$$UNIFY_{f(X,g(X,Y))}(Z) \to \mathcal{U}_{f(X,g(X,Y))}(Z, \mathcal{L}(f(\mathsf{v}, g(\mathsf{v}, \mathsf{v})), Z))$$
$$\mathcal{U}_{f(X,g(X,Y))}(f(Z, g(V, W)), true) \to \mathcal{L}(Z, V) \wedge true$$

□

We are now ready to provide the formal definition of $\mathcal{E}_{\mathrm{RT}}$: its explanation will be given soon afterwards.

Definition 5.6 (Transformation $\mathcal{E}_{\mathrm{RT}}$)
The transformation $\mathcal{E}_{\mathrm{RT}}(P)$ of a regularly typed logic program P is defined this way.

1) For every predicate $p \in \Sigma$, take the definition of p in P:

$$\begin{cases} p(\bar{t}_0^{(1)}; \bar{s}_{n_1+1}^{(1)}) \leftarrow \cdots & (C_1) \\ \vdots & \vdots \\ p(\bar{t}_0^{(k)}; \bar{s}_{n_k+1}^{(k)}) \leftarrow \cdots & (C_k) \end{cases}$$

Then produce the following rewrite rules ($i = 1..k$), plus the corresponding unification engines:

$$p[\mathcal{M}X_1,..,\mathcal{M}X_{\mathrm{in}(p)}] \to \mathcal{B}\,\langle ENC_p^{(1)}[\mathcal{M}X_1,..,\mathcal{M}X_{\mathrm{in}(p)}],..,ENC_p^{(k)}[\mathcal{M}X_1,..,\mathcal{M}X_{\mathrm{in}(p)}]\rangle$$
$$\mathcal{B}\,ENC_p^{(i)}[\mathcal{M}X_1,..,\mathcal{M}X_{\mathrm{in}(p)}] \to TRY_p^{(i)}([\mathcal{M}X_1,..,\mathcal{M}X_{\mathrm{in}(p)}],\, UNIFY_{[\bar{t}_0^{(i)}]}[X_1,..,X_{\mathrm{in}(p)}])$$
$$TRY_p^{(i)}([\mathcal{M}\bar{t}_0^{(i)}],\mathit{true}) \to \mathsf{FLOW}(C_i) \qquad TRY_p^{(i)}(X,\mathit{false}) \to \Diamond$$

2) For every $\lambda_{t_1}^{t_2}$ so far introduced, produce:

$$\lambda_{t_1}^{t_2} t_1 \to t_2 \qquad \lambda_{t_1}^{t_2}\Diamond \to \Diamond \qquad \mathcal{B}\,\lambda_{t_1}^{t_2} X \to \lambda_{t_1}^{t_2}\mathcal{B}\,X$$
$$\lambda_{t_1}^{t_2}\langle[\mathcal{M}X_1,..,\mathcal{M}X_{\mathrm{in}(p)}]|Y\rangle \to \langle\lambda_{t_1}^{t_2}[\mathcal{M}X_1,..,\mathcal{M}X_{\mathrm{in}(p)}]|\lambda_{t_1}^{t_2}Y\rangle \qquad (p \in \Sigma)$$

3) Finally, produce:

$$\mathcal{B}\,\langle X|Y\rangle \to \langle\mathcal{B}\,X|Y\rangle \qquad \mathcal{B}\,\Diamond \to \Diamond$$
$$\mathcal{B}\,\mathcal{B}\,X \to \mathcal{B}\,X \qquad \langle\Diamond|Y\rangle \to \mathcal{B}\,Y$$ □

Observe that in Point 1 in case p is not defined in P, i.e. $k = 0$, the transformation simply produces $p[\mathcal{M}X_1,..,\mathcal{M}X_{\mathrm{in}(p)}] \to \mathcal{B}\,\Diamond$.

The behaviour of $\mathcal{E}_{\mathrm{RT}}$ can be intuitively illustrated as follows.

We said earlier every clause defining a predicate p provides a way to calculate p applied to its input values. In Point 1 of the transformation the first rule says that in order to calculate p we have at our disposal the definition given in the first clause (encoded via $ENC_p^{(1)}(\cdots)$), till that in the last clause ($ENC_p^{(k)}(\cdots)$). All these different choices are grouped together, in left to right order, using a $\Diamond$-list.

The $\mathcal{B}$ symbol present in the rule before this $\Diamond$-list represents the 'backtracking command' which activates a computation.

This backtracking command can penetrate into the possibly complicated structures it encounters, via the rules (produced in Points 2 and 3 respectively) $\mathcal{B}\,\lambda_{t_1}^{t_2}X \to \lambda_{t_1}^{t_2}\mathcal{B}\,X$ and $\mathcal{B}\,\langle X|Y\rangle \to \langle\mathcal{B}\,X|Y\rangle$.

Also, the backtracking command is idempotent (rule $\mathcal{B}\,\mathcal{B}\,X \to \mathcal{B}\,X$).

As soon as $\mathcal{B}$ finds an ENC operator (encoding a certain clause), it tries to activate it via the second rules produced in Point 1: It must be checked that the (representation in the TRS of the) selected atom in the goal and the (representation of the) head of the clause unify, and this is performed via the test $UNIFY_{[\bar{t}_0^{(i)}]}[X_1,..,X_{\mathrm{in}(p)}]$.

In case the test succeeds, the rule $TRY_p^{(i)}([\mathcal{M}\bar{t}_0^{(i)}],\mathit{true}) \to \mathsf{FLOW}(C_i)$ applies the clause; in case it does not, the rule $TRY_p^{(i)}(X,\mathit{false}) \to \Diamond$ says that no result (i.e. $\Diamond$) has been produced.

The rule (produced in Point 3) $\langle\Diamond|Y\rangle \to \mathcal{B}\,Y$ says that whenever in a group of choices (contained in a $\Diamond$-list) the first argument produced no results (i.e. $\Diamond$), then it is discarded and another 'backtracking command' $\mathcal{B}$ is generated and applied to the remaining choices ($\mathcal{B}\,Y$). Note that if, instead, a result is produced, no backtracking command is generated, and so the execution stops.

Eventually, if $\mathcal{B}$ finds no results it gives no results as well (rule $\mathcal{B}\,\Diamond \to \Diamond$).

The last thing that remains to consider is the behaviour of the $\lambda_{t_1}^{t_2}$ operators.

As said at the beginning of the section, $\lambda_{t_1}^{t_2}$ is supposed to act roughly like the function $\lambda t_1.t_2$: this is expressed by the rule $\lambda_{t_1}^{t_2}t_1 \to t_2$. The difference is that it has also to deal with the other kinds of structures that can crop up: in case it finds

no results, it produces no results (rule $\lambda^{t_2}_{t_1}\Diamond \to \Diamond$), and in case it finds more choices (grouped in a $\Diamond$-list), it applies itself to all of them via the last rules produced in Point 2.

Observation 5.7 An useful shorthand is to consider only *atomic* goals, i.e. goals of the form $G = \ \leftarrow p(\bar{s};\bar{t})$. This way we can simply define the translation $\mathcal{E}_{\rm RT}(G)$ of the goal as $p[\mathcal{M}\bar{s}]$ (hence without using the convention of Subsection 2.2). From now on, for brevity, we will only consider examples with atomic goals. As an aside, note that in general this is not restrictive since, e.g., a regularly typed goal of the form $\leftarrow p_1(\bar{s}_1;\bar{t}_1),..,p_n(\bar{s}_n;\bar{t}_n)$ can be split into a goal $\leftarrow p(\bar{s}_1;\bar{t}_1,..,\bar{t}_n)$ and a clause $p(\bar{s}_1;\bar{t}_1,..,\bar{t}_n) \leftarrow p_1(\bar{s}_1;\bar{t}_1),..,p_n(\bar{s}_n;\bar{t}_n)$ (where p is a new predicate) that are both regularly typed, giving an equivalent program. □

Example 5.8 Consider the program defining the integers ([29]):

$$int(0) \leftarrow \qquad\qquad int(s(X)) \leftarrow int(X)$$

and the goal $\leftarrow int(X)$ (where the moding/typing is $int(out\!:\!Any)$).

Its translation via $\mathcal{E}_{\rm RT}$ is $(i = 1, 2)$:

$$int[\,] \to \mathcal{B}\ \langle ENC^{(1)}_{int}[\,], ENC^{(2)}_{int}[\,]\rangle$$

$$\mathcal{B}\ ENC^{(i)}_{int}[\,] \to\ TRY^{(i)}_{int}([\,], UNIFY_{[\,]}[\,]) \qquad TRY^{(1)}_{int}([\,], true) \to \lambda^{[\mathcal{M}0]}_{[\,]}[\,]$$
$$TRY^{(2)}_{int}([\,], true) \to \lambda^{[\mathcal{M}s(X)]}_{[\mathcal{M}X]}\,\lambda^{int[\,]}_{[\,]}[\,] \qquad TRY^{(i)}_{int}(X, false) \to \Diamond$$

and the term $int[\,]$ (plus the rules of the unification engine and of steps 2 and 3 of the $\mathcal{E}_{\rm RT}$ Definition). The corresponding reduction of the term in the TRS is:

$$int[\,] \to \mathcal{B}\ \langle ENC^{(1)}_{int}[\,], ENC^{(2)}_{int}[\,]\rangle \to^* \langle TRY^{(1)}_{int}([\,], UNIFY_{[\,]}[\,])), ENC^{(2)}_{int}[\,]\rangle \to^*$$
$$\langle TRY^{(1)}_{int}([\,], true), ENC^{(2)}_{int}[\,]\rangle \to \langle \lambda^{[\mathcal{M}0]}_{[\,]}[\,], ENC^{(2)}_{int}[\,]\rangle \to \langle [\mathcal{M}0], ENC^{(2)}_{int}[\,]\rangle \qquad \square$$

The TRSs produced by $\mathcal{E}_{\rm RT}$ have a quite regular structure:

Lemma 5.9 *For every regularly typed program P, $\mathcal{E}_{\rm RT}(P)$ is weakly confluent. If $\mathcal{E}_{\rm RT}(P)$ is terminating, then it is also confluent.*

We now state what existential termination properties $\mathcal{E}_{\rm RT}$ enjoys:

Theorem 5.10 *Let P and G be respectively a regularly typed program and goal: then P existentially terminates w.r.t. G iff $\mathcal{E}_{\rm RT}(P)$ terminates w.r.t. $\mathcal{E}_{\rm RT}(G)$.*

Theorem 5.11 *Let P be a regularly typed program: then P strongly existentially terminates iff $\mathcal{E}_{\rm RT}(P)$ terminates.*

Hence via the above two theorems we obtain a characterization of existential termination for the class of regularly typed programs.

Example 5.12 Graph structures are used in many applications, such as representing relations, situations or problems. Consider the program *CONNECTED*, that finds whether two nodes in a graph are connected:

$$connected(X,Y) \leftarrow arc(X,Y) \qquad connected(X,Y) \leftarrow arc(X,Z), connected(Z,Y)$$

with moding/typing $connected(in\!:\!Ground, out\!:\!Ground)$, $arc(in\!:\!Ground, out\!:\!Ground)$. Suppose the graph G is defined via the facts

$$arc(a,b) \leftarrow \qquad\qquad arc(b,c) \leftarrow \qquad\qquad arc(c,a) \leftarrow$$

When the graph is cyclic (like in this case), the program $CONNECTED \cup G$ does not strongly universally terminate. However, using Theorem 5.11, we can prove that it is strongly existentially terminating. □

Example 5.13 Reconsider the integer program of Example 5.8. This program does not strongly universally terminates, as it is trivial to see. However, the obtained TRS can be proven to be terminating, hence showing, via Theorem 5.11, that the integer program strongly existentially terminates. □

6 From ST to RT

In this section we show how to extend the previous results to the whole class of safely typed programs, using a transformation which is of independent interest.

Given a safely typed clause $C = p_0(\bar{t}_0; \bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1; \bar{t}_1), .., p_n(\bar{s}_n; \bar{t}_n)$, define $\mu(C)$ as the number of $\bar{t}_i$'s that do not satisfy the RT condition. Thus, $\mu(C)$ is somehow a measure of how much of C does not belong to RT, viz. how many atoms in a clause are 'bad' ones (note that $\mu(C) = 0$ iff $C \in \mathrm{RT}$).

Extend μ to a program P in the obvious way: $\mu(P) = \sum_{C \in P} \mu(C)$.

Now we can define a transformation $\mathcal{C}$ that translates a safely typed program into a regularly typed one.

Definition 6.1 (Transformation $\mathcal{C}$)

Let P be a safely typed program. If P is already regularly typed, then $\mathcal{C}$ leaves it unchanged ($\mathcal{C}(P) = P$).

So, suppose that P is not RT, i.e. that $\mu(P) > 0$. Take a clause C of P with $\mu(C) > 0$:

$$C = \; p_0(\bar{t}_0; \bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1; \bar{t}_1), .., p_n(\bar{s}_n; \bar{t}_n)$$

Take an $i > 0$ such that $\bar{t}_i$ makes the RT condition fail (i.e. $p_i(\bar{s}_i; \bar{t}_i)$ is a 'bad' atom of the body). Then, replace C with the following two clauses:

$$\begin{aligned} p_0(\bar{t}_0; \bar{s}_{n+1}) \leftarrow\; & p_1(\bar{s}_1; \bar{t}_1), .., p_{i-1}(\bar{s}_{i-1}; \bar{t}_{i-1}), \\ & p_i(\bar{s}_i; X_1, .., X_{\mathrm{out}(p_i)}), \\ & \mathrm{EQ}_{C,p_i}\big(X_1, .., X_{\mathrm{out}(p_i)}, \mathrm{Var}(\bar{t}_i) \cap \textstyle\bigcup_{j=0}^{i-1} \mathrm{Var}(\bar{t}_j); \mathrm{Var}(\bar{t}_i) \setminus \bigcup_{j=0}^{i-1} \mathrm{Var}(\bar{t}_j)\big), \\ & p_{i+1}(\bar{s}_{i+1}; \bar{t}_{i+1}), .., p_n(\bar{s}_n; \bar{t}_n) \end{aligned}$$

$$\mathrm{EQ}_{C,p_i}\left(\bar{t}_i, \mathrm{Var}(\bar{t}_i) \cap \textstyle\bigcup_{j=0}^{i-1} \mathrm{Var}(\bar{t}_j); \mathrm{Var}(\bar{t}_i) \setminus \bigcup_{j=0}^{i-1} \mathrm{Var}(\bar{t}_j)\right) \leftarrow$$

where $X_1, .., X_{\mathrm{out}(p_i)}$ are fresh variables and EQ_{C,p_i} is a new predicate symbol: note its mode and type is given implicitly by the above clauses.

It is not difficult to prove that this new program P' so obtained is still safely typed, and moreover $\mu(P') = \mu(P) - 1$.

Hence, repeating this process, we finally get a program Q with $\mu(Q) = 0$ (therefore regularly typed), and let $\mathcal{C}(P) = Q$. □

The intuition is that we patch the bad atoms in a program: if $p_i(\bar{s}_i; \bar{t}_i)$ is bad, we force it back to RT by inserting in place of $\bar{t}_i$ new fresh variables: next we check that these variables have been instantiated to something unifiable with $\bar{t}_i$ via the introduction of the new EQ predicate.

Example 6.2 Take the *QUICKSORTDL* program using difference lists (after [29, page 244]):

C_1 $\quad quicksort(Xs, Ys) \leftarrow quicksort\_dl(Xs, Ys, [\,])$

C_2 $\quad quicksort\_dl([X|Xs], Ys, Zs) \leftarrow partition(Xs, X, Littles, Bigs), quicksort\_dl(Littles, Ys, [X|Ys1]), quicksort\_dl(Bigs, Ys1, Zs)$

C_3 $\quad quicksort\_dl([\,], Xs, Xs) \leftarrow$

(plus the rules for *partition*), with moding/typing $quicksort(in\colon NatList, in\colon NatList)$, $quicksort\_dl(in\colon NatList, in\colon NatList, out\colon NatList)$, $partition(in\colon NatList, in\colon Nat, out\colon$

$NatList, out{:}NatList$). This program is safely typed but not regularly typed because of the first and second clause: the atom $quicksort\_dl(Xs, Ys, [\,])$ in C_1 and the atom $quicksort\_dl(Littles, Ys, [X|Ys1])$ in C_2 are the only 'bad' ones ($\mu(QUICKSORTDL) = 2$). Applying $\mathcal{C}$, we obtain in place of C_1 and C_2 the clauses:

C_1' $quicksort(Xs, Ys) \leftarrow quicksort\_dl(Xs, Ys, X_1), \mathrm{EQ}_1(X_1)$
C_1'' $\mathrm{EQ}_1([\,]) \leftarrow$
C_2' $quicksort\_dl([X|Xs], Ys, Zs) \leftarrow partition(Xs, X, Littles, Bigs), quicksort\_dl(Littles, Ys, X_1), \mathrm{EQ}_2(X_1, X, Ys1), quicksort\_dl(Bigs, Ys1, Zs)$
C_2'' $\mathrm{EQ}_2([X|Ys1], X, Ys1) \leftarrow$

where EQ_1 is moded/typed ($in{:}NatList$) and EQ_2 ($in{:}NatList, in{:}Nat, out{:}NatList$). □

Observe that the transformation $\mathcal{C}$ can in general introduce some extra computations since it delays the test on the output arguments (via EQ). However, it somehow retains the original structure of the program, since it preserves the logical meaning in the following sense:

Theorem 6.3 *Let P and G be a safely typed program and goal. Then ϑ is an SLD computed answer substitution for $\mathcal{C}(P \cup \{G\})$ iff $\vartheta|_{\mathrm{Var}(G)}$ is an SLD computed answer substitution for $P \cup \{G\}$.*

The proof of the above theorem makes use of fold/unfold techniques.

As far as termination is concerned, the following result holds:

Lemma 6.4 *Let P and G be a safely typed program and goal. For every ordinal k, if $\mathcal{C}(P)$ k-terminates w.r.t. $\mathcal{C}(G)$ then P k-terminates w.r.t. G, and if $\mathcal{C}(P)$ strongly k-terminates then P strongly k-terminates.*

Hence we can analyze the termination behaviour of a safely typed program by applying the compound transformation

$$\mathcal{E}_{\mathrm{ST}} \;=\; \mathcal{E}_{\mathrm{RT}} \circ \mathcal{C}$$

Theorem 6.5 *Let P and G be respectively a safely typed program and goal: then P existentially terminates w.r.t. G if $\mathcal{E}_{\mathrm{ST}}(P)$ terminates w.r.t. $\mathcal{E}_{\mathrm{ST}}(G)$.*

Theorem 6.6 *Let P be a safely typed program: then P strongly existentially terminates if $\mathcal{E}_{\mathrm{ST}}(P)$ terminates.*

7 The k-termination case

So far, we have presented only criteria on existential termination. In this section, we provide more general results to cope with the whole spectrum of k-termination.

Through this section, $\mathcal{A}$ and $\mathcal{S}$ denote two new fresh symbols.

Theorem 7.1 *Let P and G be a regularly typed (resp. safely typed) program and goal. Then for every k s.t. $0 < k < \omega$, P k-terminates w.r.t. G iff (resp. if) $\mathcal{E}_{\mathrm{ST}}(P) \cup \{\mathcal{A}(\mathcal{S}(X), \langle [\,]|W\rangle) \rightarrow \mathcal{A}(X, \mathcal{B}\,W),\ \mathcal{A}(\mathcal{S}(X), \langle [Y|Z]|W\rangle) \rightarrow \mathcal{A}(X, \mathcal{B}\,W)\}$ terminates w.r.t. $\mathcal{A}(\underbrace{\mathcal{S}\cdots\mathcal{S}}_{k-1}\Diamond, \mathcal{E}_{\mathrm{ST}}(G))$.*

The intuition is that we consider reductions in the TRS not of the original term $\mathcal{E}_{\mathrm{ST}}(G)$, but of the term $\mathcal{A}(\mathcal{S}\cdots\mathcal{S}\Diamond, \mathcal{E}_{\mathrm{ST}}(G))$ that 'counts' how many answers have been so far produced. The counter is stored in the first argument of $\mathcal{A}$, initially set to a unary representation of $k-1$. Each time one answer has been found, one of the two added rules defining $\mathcal{A}$ is applied, forcing a new backtracking ($\mathcal{B}\,W$) and decrementing the counter by one, till all the k answers have been found.

As far as ω-termination is concerned, it is so close to universal termination that there seems to be no way to provide a specific criterion for ω-termination: to infer ω-termination once can nevertheless use a criterion for universal termination (see later).

Example 7.2 Consider the following program *PATH* computing paths in a graph (the goal asks for paths from a_1 to b):

$\leftarrow$ *path(*a_1*,b,X)*
path(X,Y,[X,Y])$\leftarrow$ *arc(X,Y)*
path(X,Y,[X|Xs])$\leftarrow$ *arc(X,Z), path(Z,Y,Xs)*

With moding/typing $path(in : Ground, in : Ground, out : List)$, $arc(in : Ground, in : Ground)$ (in the first clause) and $arc(in : Ground, out : Ground)$ (in the second clause), it is regularly typed.

Suppose the graph G_k is defined via the facts

$arc(a_1, b) \leftarrow, .., arc(a_k, b) \leftarrow,\ arc(a_1, a_2) \leftarrow, .., arc(a_k, a_{k+1}) \leftarrow,\ arc(a_{k+1}, a_{k+1}) \leftarrow$

Using the above Theorem 7.1, we can prove that for every $0 < k < \omega$, the program $PATH \cup G_k$ is k-terminating. Note also that all these programs do not universally terminate: *$PATH \cup G_k$ is not $k+1$-terminating*. Incidentally, this also provides a proof that, unlike in the strong termination case, in the case of termination w.r.t. a goal the k-termination hierarchy does not collapse. □

Theorem 7.3 *Let P and G be a regularly typed (resp. safely typed) program and goal. Then P universally terminates w.r.t. G iff (resp. if) $\mathcal{E}_{\mathrm{ST}}(P) \cup \{\mathcal{A}(\langle[]|Z\rangle) \rightarrow \mathcal{A}(\mathcal{B}\ Z), \mathcal{A}(\langle[X|Y]|Z\rangle) \rightarrow \mathcal{A}(\mathcal{B}\ Z)\}$ terminates w.r.t. $\mathcal{A}(\mathcal{E}_{\mathrm{ST}}(G))$.*

We turn now our attention towards strong k-termination.

Since, by Theorem 4.5, strong k-termination with $1 \leq k \leq \omega$ coincides with strong existential termination, Theorem 6.6 suffices in all these cases. The only remaining case is strong universal termination:

Theorem 7.4 *Let P be a regularly typed (resp. safely typed) program. Then P strongly universally terminates iff (resp. if) $\mathcal{E}_{\mathrm{ST}}(P) \cup \{\mathcal{A}(\langle[]|Z\rangle) \rightarrow \mathcal{A}(\mathcal{B}\ Z), \mathcal{A}(\langle[X|Y]|Z\rangle) \rightarrow \mathcal{A}(\mathcal{B}\ Z)\}$ terminates.*

Example 7.5 Consider the program *QUICKSORTDL* seen in Example 6.2: via the above theorem we can prove that it is strongly universally terminating.

Analogously, we can prove for instance that the program to solve the Hanoi towers problem (cf. [29, pp. 64–65]) with moding/typing $hanoi(in : List, out : List)$, the usual quicksort program ([29, page 56]) with $quicksort(in : List, out : List)$, and the English sentences parser ([29, pp. 256–258]) with $sentence(in : Ground, out : Ground)$ are all strongly universally terminating. □

8 Normal Logic Programs

After having analyzed definite logic programming, we extend the results previously obtained to normal logic programming, that is allowing usage of *negation*. As usual in Prolog, negated atoms are solved using the negation as finite failure procedure, i.e. they succeed if and only if they finitely fail. Since we have already defined classes of definite logic programs, we can give the definition of their extensions to normal logic programs inductively on the number of negative literals:

Definition 8.1 A clause is normal safely typed iff either it is safely typed, or:
if the clause is of the form $p_0(\bar{t}_0; \bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1; \bar{t}_1), .., not(p_k(\bar{s}_k; \bar{t}_k)), .., p_n(\bar{s}_n; \bar{t}_n)$, then both both $p_0(\bar{t}_0; \bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1; \bar{t}_1), .., p_k(\bar{s}_k; \bar{t}_k)$ and $p_0(\bar{t}_0; \bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1; \bar{t}_1), .., p_{k-1}(\bar{s}_{k-1}; \bar{t}_{k-1}), p_{k+1}(\bar{s}_{k+1}; \bar{t}_{k+1}), .., p_n(\bar{s}_n; \bar{t}_n)$ are normal safely typed.
A program is *Normal Safely Typed* (NST) if each of its clauses is.
The class of *Normal Regularly Typed* (NRT) logic programs is defined analogously. □

Example 8.2 Suppose p and q have both moding/typing $(in\colon Any, out\colon Any)$. Then the clause $p(X, f(Z)) \leftarrow q(X,Y), not\,(p(Y,Z)), q(Y,Z)$ is normal regularly typed since both $p(X, f(Z)) \leftarrow q(X,Y), p(Y,Z)$ and $p(X, f(Z)) \leftarrow q(X,Y), q(Y,Z)$ are regularly typed. □

Now we have to extend the definition of $\mathcal{E}_{\mathrm{RT}}$ to cope with negation. The modification is quite simple. The definition of FLOW (cf. Def. 5.1) is extended this way: it acts like before, only that if a predicate p_i in the body of the clause is negated, i.e. of the form $not\,p_i(\cdots)$, then in the produced term it appears as the compound function $not \circ p_i$, where not is defined as follows:

$$not\,\Diamond \rightarrow \langle [\,] \rangle \qquad not\,\langle [\,] | X \rangle \rightarrow \Diamond \qquad not\,\langle [X|Y]|Z \rangle \rightarrow \Diamond$$

The explanation of these rules is perfectly natural: since $not\,p_i(\cdots)$ succeeds iff $p_i(\cdots)$ finitely fails, in the TRS we first calculate $p_i(\cdots)$ and then apply to it the not operator: if no answers are returned ($\Diamond$), it outputs a result $[\,]$ via the rule $not\,\Diamond \rightarrow \langle [\,] \rangle$ ($[\,]$ corresponds to the fact that a successful negative literal produces no bindings), whereas if a result is returned, it outputs no result (via the other two rules).

This way we obtain a new basic transformation $\mathcal{E}_{\mathrm{NRT}}$ that extends $\mathcal{E}_{\mathrm{RT}}$ from regularly typed to normal regularly typed programs. Hence, all the transformations previously defined (and their results) extend to normal logic programs, with the correspondence RT$\rightsquigarrow$NRT, and ST$\rightsquigarrow$NST.

For brevity, we only cite the cases of strong existential and universal termination: all the others are similarly obtained using the above syntactic correspondence.

Theorem 8.3 *Let P be a normal regularly typed (resp. normal safely typed) program: then P strongly existentially terminates iff (resp. if) $\mathcal{E}_{\mathrm{NST}}(P)$ terminates.*

Theorem 8.4 *Let P be a normal regularly typed (resp. normal safely typed) program. Then P strongly universally terminates iff (resp. if) $\mathcal{E}_{\mathrm{NST}}(P) \cup \{\mathcal{A}(\langle [\,] | Z \rangle) \rightarrow \mathcal{A}(\mathcal{B}\,Z), \mathcal{A}(\langle [X|Y]|Z \rangle) \rightarrow \mathcal{A}(\mathcal{B}\,Z)\}$ terminates.*

Example 8.5 Consider the following normal program

$$p \leftarrow not\ q \qquad q \leftarrow \qquad q \leftarrow q$$

Via Theorem 8.3, we can prove this program is strongly existentially terminating. Nevertheless, it is *not* universally terminating. □

Example 8.6 A much more complicated example of normal program is given by the Block-World Planner of [29, pp. 221–224], that with moding/typing $transform(in : State, in : State, out : Plan)$ ($State$ and $Plan$ are suitable types) can be proven via Theorem 8.3 to be strongly existentially terminating. □

Example 8.7 The normal program to solve efficiently the n-queens problem, after [29, page 211], moded/typed $queens(in\colon Ground, out\colon List)$, can be proven via Theorem 8.4 to be strongly universally terminating. □

9 Relations with previous work

As said, the main contribution of the paper is aimed towards the open problem of existential termination: indeed, as mentioned in the introduction, the very few works on the subject ([9, 18, 6]) give only expressibility results and are, presently, of no practical use (cf. [13]). Also, they do not cope with the intermediate degrees of k-termination ($2 \leq k \leq \omega$). Very recently, other two works have addressed the subject, namely [25, 20]. The first work [25] has introduced the concept of k-termination, and shown how it can be studied using functional programming techniques. However, the class of normal logic programs to which this analysis can be applied is rather limited, since the main goal of the work is completely different, namely to identify what part

of logic programming is just functional programming in disguise. The second work is [20], but besides do not treating negation, its practical importance is at the moment unclear.

Thus, in order to make a comparison with other works we have to *restrict* our approach to the *universal termination* case only. A first point that can be made is that our approach is able to satisfactorily cope with *negation*: the only works that manage to cover some aspects of negation are, to the best of our knowledge, very few.

In [4] a theoretical criterion (acceptable programs) is given: however, this result is considered as a main theoretical foundation, rather than an effective methodology: no practical way to automate or semi-automate the criterion is known, since it heavily relies on semantical information (e.g. it must be provided a model of the program which is also a model of the completion of its 'negative part'). Recently, a novel methodology that overcomes some of the difficulties of this method due to the use of the semantic information, has been introduced in [22].

In [30] a sufficient criterion for termination of normal logic programs is presented. This criterion suffers from the same drawback of [4]: it is far from being easily implemented being exclusively semantically-based (in addition, it requires its main semantical information to be provided by some other proof method). Also, treatment of negation is coped with by assuming that every negated literal *will always succeed*, which readily limits by far the usefulness of the approach to negation.

Another recent work is [11]: the importance of this work is that it manages to treat not only logic programming, but the whole class of (normal) constraint logic programming, even in the presence of delays. Moreover, it also provides a characterization of termination when negation is not present. A limitation is that the treatment of negation is analogous to the aforementioned [30].

Theoretically, comparing the power of all these approaches with ours gives the result that they overlap but no one is strictly more powerful than the other.

Turning to all the other works on the subject, *which do not cover negation*, we have already discussed in the introduction what are the advantages of the transformational approach towards all the other methods (for a panoramic, see [13]). Hence, it remains to ask how our approach fits w.r.t. all the other papers based on the transformational approach (cf. [26, 19, 1, 8, 23, 5]).

First, all the cited works only cover the 'strong universal termination' case.

Second, all the works (but for [23]) can only treat *well moded* programs (i.e., cf. Section 3, the class obtained from ST when the unique type allowed is *Ground*), hence restricting by far the applicability scope.

Third, call a transformation $\mathcal{T}_1$ *at least as powerful as* $\mathcal{T}_2$ (notation $\mathcal{T}_1 \geq \mathcal{T}_2$) if, for every logic program P, $\mathcal{T}_2(P)$ terminates implies that $\mathcal{T}_1(P)$ terminates (i.e., every program that can be proven terminating by $\mathcal{T}_2$ can be proven terminating even by $\mathcal{T}_1$). Call $\mathcal{T}_1$ *strictly more powerful than* $\mathcal{T}_2$ if $\mathcal{T}_1 \geq \mathcal{T}_2$ and $\mathcal{T}_1 \not\leq \mathcal{T}_2$. Then, with the exception of one of the two transformations of [23] ($\mathrm{T}_{\mathrm{fwm}}$), which seems to be only of theoretical interest, we have the following result:

Theorem 9.1 *Even when restricting to a type system with the only type Ground, our transformation is strictly more powerful than all the transformations in [26, 19, 1, 8, 23, 5].*

References

[1] G. Aguzzi and U. Modigliani. Proving termination of logic programs by transforming them into equivalent term rewriting systems. *FST&TCS*, LNCS 761, pp. 114–124. 1993.

[2] A. Aiken and T.K. Lakshman. Directional type checking of logic programs. In *Proc. of SAS*, LNCS 864, pages 43–60, 1994.

[3] K.R. Apt and S. Etalle. On the unification free Prolog programs. *Proc. MFCS*, LNCS 711, pp. 1–19. Springer, 1993.

[4] K.R. Apt and D. Pedreschi. Proving termination of general Prolog programs. In *Proc. TACS*, LNCS 526, pages 265–289. Springer–Verlag, 1991.

[5] T. Arts and H. Zantema. Termination of logic programs using semantic unification. In *Proc. 5th LoPSTr*, LNCS. 1996. To appear.

[6] M. Baudinet. Proving termination properties of Prolog programs: a semantic approach. *Journal of Logic Programming*, 14:1–29, 1992.

[7] F. Bronsard, T.K. Lakshman, and U.S Reddy. A framework of directionality for proving termination of logic programs. *Proc. JICSLP*, pp. 321–335. The MIT Press, 1992.

[8] M. Chtourou and M. Rusinowitch. Méthode transformationnelle pour la preuve de terminaison des programmes logiques. Manuscript, 1993.

[9] K.L. Clark and S.-Å. Tärnlund. A first order theory of data and programs. In B. Gilchrist, editor, *Proc. IFIP Congress on Information Processing*, pp. 939–944, 1977.

[10] H. Coelho and J.C. Cotta. *Prolog by Example*. Springer–Verlag, 1988.

[11] L. Colussi, E. Marchiori, and M. Marchiori. On termination of constraint logic programs. In *Proc. CP'95*, LNCS 976, pp. 431–448. Springer, 1995.

[12] T. Conway, F. Henderson, and Z. Somogyi. Code generation for Mercury. In *Proc. International Logic Programming Symposium*, pp. 242–256. The MIT Press, 1995.

[13] D. de Schreye and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19,20:199–260, 1994.

[14] S.K. Debray. Static inference of modes and data dependencies in logic programs. *ACM TOPLAS*, 11(3):418–450, July 1989.

[15] N. Dershowitz. Termination of rewriting. *J. of Symbolic Computation*, 3:69–116, 1987.

[16] N. Dershowitz and C. Hoot. Topics in termination. In *Proceedings of the Fifth RTA*, LNCS 690, pp. 198–212. Springer, 1993.

[17] N. Dershowitz and J. Jouannaud. Rewrite systems. *Handbook of Theoretical Computer Science*, vol. B, chapter 6, pp. 243–320. Elsevier – MIT Press, 1990.

[18] N. Franchez, O. Grumberg, S. Katz, and A. Pnueli. Proving termination of Prolog programs. In R. Parikh, editor, *Logics of programs*, pp. 89–105. Springer, 1985.

[19] H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. *CTRS'92*, LNCS 656, pp. 216–222. Springer, 1993.

[20] G. Levi and F. Scozzari. Contributions to a theory of existential termination for definite logic programs. *GULP-PRODE'95*, pp. 631–642. 1995.

[21] J.W. Lloyd. *Foundations of Logic Programming*. Springer–Verlag, second edition, 1987.

[22] E. Marchiori. Practical methods for proving termination of general logic programs. *The Journal of Artificial Intelligence Research*, 1996. To appear.

[23] M. Marchiori. Logic programs as term rewriting systems. *Proc. 3rd Int. Conf. on Algebraic and Logic Programming*, LNCS 850, pp. 223–241. Springer, 1994.

[24] M. Marchiori. Localizations of unification freedom through matching directions. *Proc. International Logic Programming Symposium*, pp. 392–406. The MIT Press, 1994.

[25] M. Marchiori. The functional side of logic programming. In *Proc. ACM FPCA*, pages 55–65. ACM Press, 1995.

[26] K. Rao, D. Kapur, and R.K. Shyamasundar. A transformational methodology for proving termination of logic programs. *5th CSL*, LNCS 626, pp. 213–226, Springer, 1992.

[27] K. Rao, P.K. Pandya, and R.K. Shyamasunder. Verification tools in the development of provably correct compilers. FME'93, LNCS 670, pp. 442–461. Springer, 1993.

[28] Y. Rouzaud and L. Nguyen-Phoung. Integrating modes and subtypes into a Prolog type checker. In *Proc. JICSLP*, pages 85–97. The MIT Press, 1992.

[29] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1986.

[30] B. Wang and R.K. Shyamasundar. A methodology for proving termination of logic programs. *Journal of Logic Programming*, 21(1):1–30, 1994.