Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 248 (2009) 93–113
www.elsevier.com/locate/entcs

Methods for Proving Termination of Rewriting-based Programming Languages by Transformation

Francisco Durán[1]

DLCC, Universidad de Málaga, Málaga, Spain

Salvador Lucas[2]

DSIC, Universidad Politécnica de Valencia, Valencia, Spain

José Meseguer[3]

CS Dept., University of Illinois at Urbana-Champaign, Urbana, IL, USA

Abstract

Despite the remarkable development of the theory of termination of rewriting, its application to high-level (rewriting-based) programming languages is far from being optimal. This is due to the need for features such as conditional equations and rules, types and subtypes, (possibly programmable) strategies for controlling the execution, matching modulo axioms, and so on, that are used in many programs and tend to place such programs outside the scope of current termination tools. The operational meaning of such features is often formalized in a proof theoretic manner by means of an inference system rather than just by a rewriting relation. The corresponding termination notions can also differ from the standard ones. During the last years we have introduced and implemented different notions and transformation techniques which have been proved useful for proving and disproving termination of such programs by using existing tools for proving termination of (variants of) rewriting. In this paper we provide an overview of our main contributions.

Keywords: Program Analysis and Verification, Rewriting Logic, Term Rewriting, Termination, Tools

1 Programs and logics

Rewriting-based languages with expressive features are supported by expressive *logics*, that typically include less expressive ones as sublogics. In this regard, member-

[1] Partially supported by the EU (FEDER) and Spanish MEC/MICINN under grants TIN2005-09405-C02-01 and TIN2008-03107. Email: duran@lcc.uma.es

[2] Partially supported by the EU (FEDER) and the Spanish MEC/MICINN, under grant TIN 2007-68093-C02-02. Email: slucas@dsic.upv.es

[3] Partially supported by ONR grant N00014-02-1-0715. Email: meseguer@cs.uiuc.edu

doi:10.1016/j.entcs.2009.07.062

ship equational logic (MEL) [29,3] has proved to be a very expressive *logical framework*, in which a wide range of partial and total equational logics can be faithfully embedded [29]. In particular, MAUDE's equational sublanguage, whose (functional) modules are membership equational theories (enriched with some *context-sensitivity* information regarding the possibility of performing reductions within the arguments of the function calls, see [21,22]), has itself a simple representation into this framework.

Example 1.1 Consider the following Maude functional module [8]:

```
fmod LengthOfFiniteListsAndTake is
   sorts Nat NatList NatIList . subsort NatList < NatIList .
   op 0 : -> Nat .
   op s : Nat -> Nat .
   op zeros : -> NatIList .
   op nil : -> NatList .
   op cons : Nat NatIList -> NatIList [strat (1 0)] .
   op cons : Nat NatList -> NatList [strat (1 0)] .
   op take : Nat NatIList -> NatList .
   op length : NatList -> Nat .
   vars M N : Nat .
   var  IL : NatIList .
   var  L : NatList .
   eq zeros = cons(0,zeros) .
   eq take(0, IL) = nil .
   eq take(s(M), cons(N, IL)) = cons(N, take(M, IL)) .
   eq length(nil) = 0 .
   eq length(cons(N, L)) = s(length(L)) .
endfm
```

where sorts `NatList` and `NatIList` are intended to classify finite and infinite lists of natural numbers, respectively. The function `zeros` generates an infinite list of zeros, and `take` can be used to obtain an initial segment of a list by giving the number of items we want to extract. Finally, `length` computes the length of a *finite* list. Note the *overloaded* operator `cons`, which can be used for building both finite and infinite lists of natural numbers and is declared with evaluation *strategy*[4] (1 0). The interpretation of this strategy annotation is as follows: the evaluation of an expression `cons(`h`,`t`)` proceeds by first evaluating h and then trying a reduction step at the top position (represented by 0). No evaluation is allowed on the second argument t because index 2 is missing from the annotation. Note also that `NatList` is a subsort of `NatIList`, thus allowing the use of `take` to extract finite sublists of items both from finite and *infinite* lists.

With MEL, complex types can be described by means of explicit *memberships* which establish whether a given (instance of an) expression belongs to a given sort.

Example 1.2 The following *palindrome recognizer program* `PALINDROME` is a membership equational program expressible in MAUDE as follows [11]:

```
fmod PALINDROME is
  protecting QID .  *** Imports sort Qid (quoted identifiers)
  sorts List Pal .
  subsorts Qid < Pal < List .
  op nil : -> Pal .
  op __ : List List -> List [assoc id: nil] .
```

[4] Actually, the final 0 could be removed from the strategy annotation for `cons` because no rule applies on top of terms having `cons` as root symbol. However, since zero-ended strategy annotations are usually assumed/required in OBJ/Maude programs (see, e.g., [12]), we keep it in our example.

```
  var I : Qid .
  var P : Pal .
  mb I P I : Pal .  *** membership axiom
endfm
```

This program (where list concatenation is expressed with empty syntax and satisfies associativity (`assoc`) and identity (`id` for `nil`) axioms) is terminating, that is, given a list of quoted identifiers the specification can always be used to compute in a finite number of steps whether it is a palindrome, i.e., has sort `Pal`, or not. But note that no rewriting at all is involved.

In MEL, memberships can also be conditional, as in the following example:

Example 1.3 The following functional module

```
fmod INF is
  sorts Nat Inf .
  subsort Inf < Nat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  var N : Nat .
  cmb s(N) : Inf if s(s(N)) : Inf .
endfm
```

provides an interesting example of a *nonterminating* program involving no rewrite rule (borrowed from [11, Introduction]). Here, a conditional membership establishes that terms `s(N)` (for terms `N` of sort `Nat`) have sort `Inf` provided that `s(s(N))` has sort `Inf` too. Again, no rewritings are specified here.

Generalized Rewrite Theories (GRT) [4] are a recent generalization of rewrite theories at the heart of the most recent formulation of Maude [5]. In contrast to MEL, which only covers the *functional* modules of Maude, GRT cover the most general of Maude modules, namely, *system* modules. In contrast to a MEL theory, a rewrite theory $\mathcal{R}$ (and therefore a Maude system module) contains both equations E and rewrite rules R. Both equations and rules are computed by rewriting (perhaps modulo some structural axioms A). But the equations E (including memberships!) and the rules R have a different mathematical and operational semantics. In particular, equations in E can be conditional, but their conditions can only involve other equational axioms. Instead, a conditional rule in R can have both equational conditions and non-equational rewrite conditions. This means that there are *two* different rewrite relations, $\rightarrow_E$ and $\rightarrow_R$. It also means that termination may crucially depend on the distinction between $\rightarrow_E$ and $\rightarrow_R$. We can illustrate this crucial distinction between equations E and rules R with the following simple example.

Example 1.4 Consider the following system module [10]:

```
mod MARKS-LISTS is
  sorts Nat List MNat MList .
  subsort List < MList .
  subsort Nat < MNat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op # : -> MNat .
  op nil : -> List .
  op _;_ : Nat List -> List .
  op _;_ : MNat MList -> MList .
  op <_> : MList -> MList .
  vars M N N1 N2 N3 : Nat .
  vars L L' : List .
  vars X : MNat .
```

```
  vars XS : MList .
  crl [introduce] : < L > => < # ; L > if < N1 ; N2 ; N3 ; L' > := < L > .
  rl  [propagate] : # ; (N ; M ; L) => N ; (# ; M ; L) .
  rl  [remove] : # ; N ; L => L .
endm
```

which given a list representation of a multiset of natural numbers (nondeterministically) computes its submultisets of size 2. A mark '`#`' is *introduced* into a given `List` of numbers (of sort `Nat`) to yield a *marked* list of sort `MList` (supersort of `List`). The matching condition `< N1 ; N2 ; N3 ; L' > := < L >` in the conditional rule ensures that '`#`' is introduced into lists of at least three elements. Note that *no equation is specified*, i.e., $E = \emptyset$ and $\mathcal{R}$ consists of the three rules `introduce`, `propagate`, and `remove`. As we discuss below, this fact is essential to appropriately explain the termination behavior of the program. Symbol `#` is intended to mark a number to be *removed* by using the third rule (thus producing a *sublist* of the original one). The mark can be *propagated* inside the structure of the list until it is finally *removed* (together with its companion number) to produce a list of sort `List` on which we can restart the process. Objects from both `List` and `MList` can be built by using a single *overloaded* constructor `_;_`.

2 Termination of rewriting-based programs

Termination has been studied in depth in the abstract framework of rewrite systems [1,32,35]. There are many available tools for proving termination of (different variants of) rewrite systems (e.g., AProVE [14], C*i*ME [7], MU-TERM [23], TPA [20], TTT [18],...). The notions coming from the already quite mature theory of termination of Term Rewriting Systems (TRSs) provide a basic collection of abstractions, notions, and methods for treating termination problems in sophisticated programming languages. A suitable way to prove termination of programs written in declarative programming languages like CafeOBJ [13], ELAN [2], Haskell [19], Maude/OBJ [5,17], or Prolog [31] is translating them into (variants of) TRSs and then using techniques and tools for proving termination of rewriting, see [11,15,24,34] for recent proposals of concrete procedures and tools that apply to the aforementioned programming languages.

In rewriting-based programming languages like CafeOBJ, ELAN, or Maude, one is often tempted to map termination problems for programs in such languages directly into termination problems for TRSs or *conditional* TRSs (CTRSs, see [32] for a good and sufficiently updated account of notions and results in this subfield) in quite a straightforward way. However, handling programs in this way can often lead to wrong conclusions about their real termination behavior. This is because the programs make use of additional features whose appropriate consideration is often essential to prove termination and which are not captured by the computational model of (pure) term rewriting:

(i) Sorts, subsorts, and operator overloading, as in Examples 1.1 and 1.4.

(ii) Memberships, as in Example 1.2, and conditional memberships, as in Example 1.3.

(iii) Conditions, which may introduce extra variables, as in Example 1.4.

(iv) Matching conditions (modulo a set of equations) in the conditional part of rules, as in Example 1.4.

(v) Mixed rewriting, membership, and matching conditions in the conditional part of the rules.

(vi) Context-sensitivity, which permits the introduction of annotations to specify the arguments which can be evaluated in each function call (as in the program in Example 1.1 for the two overloaded versions of `cons`).

(vii) Fixed evaluation strategies (e.g., leftmost-innermost or leftmost-outermost); for instance, the Maude programs in the examples above use a default *leftmost-innermost* strategy.

(viii) Programmable evaluation strategies, which specify a particular ordering for the evaluation of the arguments in function calls [12]: a typical example is the strategy `(1 0 2 3)` associated to the symbol `if_then_else_fi`.

(ix) Rewriting modulo axioms like associativity (A), commutativity (C), identity (I), AC, ACI, and so on, as in Example 1.2 (where the 'empty-syntax' concatenation of lists is an associative operator).

Let us briefly illustrate the role of some of these features in determining the termination behavior of a program with some discussion concerning the examples above:

(i) Modeling `MARKS-LISTS` in Example 1.4 as a CTRS yields a *nonterminating* system: the matching condition is translated into a rewriting condition which becomes part of the obtained conditional rule

$$\texttt{< L >} \rightarrow \texttt{< \# ; L >} \text{ if } \texttt{< L >} \rightarrow \texttt{< N1 ; N2 ; N3 ; L' >}$$

The application of this rule requires the reduction of (an instance of) `< L >` into (an instance of) `< N1 ; N2 ; N3 ; L' >` to satisfy the condition. Since the left-hand side `< L >` of the conditional rule itself can also be considered in any attempt to satisfy the conditional part of the rule, we run into a nonterminating computation, see [25] for a deeper discussion on this issue.

However, viewed as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and executed as a Maude program, `MARKS-LISTS` is terminating. The key point here is that solving the matching condition involves *no rewriting step*. Matching conditions are evaluated in Maude with respect to the set E of *equations* which is different from the set of rules R in $\mathcal{R}$. A *matching-modulo-E* semantics is given for solving matching conditions. In our `MARKS-LISTS` example, E is empty and the matching condition becomes *syntactic pattern matching*. *No reduction* is allowed! Indeed, only when the *two* kinds of E- and R-computations which are implicit in the specification are (separately!) taken into account, are we able to prove this program terminating.

(ii) Sort information (including both the existence of a *sort hierarchy* as the one which has been specified in `LengthOfFiniteListsAndTake` and `MARKS-LISTS`

and also the association of a *sort discipline* to the arguments of symbols and terms built from them), context-sensitivity, etc., can play a crucial role in the termination behavior and hence in any attempt to provide an automatic proof of it. For instance, `LengthOfFiniteListsAndTake` is terminating. However,

(a) If we disregard sort information, a nonterminating *context-sensitive* TRS (CS-TRS [5] [21,22]) is obtained, as shown by the infinite rewrite sequence:

$$\texttt{length(}\underline{\texttt{zeros}}\texttt{)} \to \underline{\texttt{length(cons(0,zeros))}} \to \texttt{s(}\underline{\texttt{length(zeros)}}\texttt{)} \to \cdots$$

(b) If we disregard context-sensitivity information (thus enabling reduction in the second argument of `cons`), then $\underline{\texttt{zeros}} \to \texttt{cons(0,}\underline{\texttt{zeros}}\texttt{)} \to \cdots$

(iii) Even though no rewriting is involved in any computation with program `INF` above (specifying only a *conditional membership* whose conditional part is a membership again), this program is *nonterminating* (as one can easily check by using the MAUDE interpreter).

(iv) The following program, involving both equations and memberships, shows how the recursive interaction between rewriting and membership computations can lead to subtle nontermination problems:

```
fmod INF2 is
   sorts S .
   op a : -> [S] .
   op f : [S] -> [S] [strat (0)] .
   ceq a = f(a) if a : S .
endfm
```

Note that both `a` and `f` do not have a sort, and are only defined at the *kind* level, using the kind `[S]` associated to the sort `S` (see Section 4.2). Note also that `f` has a strategy `(0)`, forbidding reductions in the argument of `f`. MAUDE fails to terminate when trying to reduce the term `a`. The problem is that the computation of the membership `a:S` requires the reduction of `a`. This leads to an infinite computation (see below).

What these examples show, most strikingly the `PALINDROME`, `INF`, and `INF2` specifications, is that termination of a declarative program may not involve rewriting at all, or, as in the case of `INF2`, may involve *both* rewriting and other computational relations. Thus, the standard (rewriting-based) termination notions that have been developed for rewriting-based programming languages, including those for CTRSs, are insufficient for dealing with termination of MEL or rewriting logic programs. For this reason, we use in this paper a proof-theoretic termination notion, called *operational termination* [25]. This notion is *parametric* on the logic: it can be defined not just for MEL, but for many other logics, that may or may not involve rewriting in their computations. Intuitively, a program is operationally terminating if all its well-formed proof trees are finite. For example, the nontermination of the

[5] A CS-TRS $(\mathcal{R}, \mu)$ is a TRS $\mathcal{R}$ together with a replacement map μ, i.e., a mapping from symbols f into sets of their argument indices which specifies where reductions are allowed.

INF program is witnessed by the infinite proof tree,

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\texttt{s(s(s(N))):Inf}}}{\texttt{s(s(N)):Inf}}}{\texttt{s(N):Inf}}}$$

Similarly, an attempt to evaluate a w.r.t. INF2 above leads to the infinite proof tree

$$\dfrac{\dfrac{\dfrac{\cdots}{\texttt{a}\rightarrow\texttt{f(a)}} \quad \texttt{f(a):s}}{\texttt{a:s}}}{\texttt{a}\rightarrow\texttt{f(a)}}$$

showing that INF2 fails to be *operationally terminating*.

As we further explain in Section 3, one key advantage of the notion of operational termination is that it is parametric on the logic underlying the given programming language. In particular, it is useful to clarify termination issues for *conditional* specifications, even for the special case of term rewriting specifications [25]. Intuitively, and this is for example illustrated by INF2 above, the problem is that a conditional specification may have a terminating rewriting relation (INF2 does, since it is the empty relation) and still be nonterminating by "looping" in evaluating a condition. Where some notions of conditional termination run aground, for example that of "effective termination" (see [25]), is in failing to give a proper account of such looping. In operational termination terms, any nonterminating behavior, either in the rewrite relation, or in a condition, or in any other computational relation, is both detected and characterized by the existence of an infinite proof tree.

3 Operational termination

We consider a logic $\mathcal{L}$ defined by inference rules, parameterized by a *theory* $\mathcal{S}$. That is, we focus on provability, and assume the axiomatic framework of general logics [28], in which what we call a *logic* becomes a particular style of presenting an *entailment system*. We refer to [4] for a more detailed account of the axiomatic metalogical background that we assume in what follows. The notion of *operational termination* [25] is *parametric* on the inference system. We briefly recall the notions we need for our purpose.

Definition 3.1 The set of (finite) proof trees for a theory $\mathcal{S}$ in a logic $\mathcal{L}$ and the head of a proof tree are defined inductively as follows. A *proof tree* is

- either an *open goal*, simply denoted as φ, where φ is a formula for $\mathcal{S}$; then, we define $head(\varphi) = \varphi$.

- or a *non-atomic* tree with φ as its head, denoted as

$$\frac{T_1 \quad \cdots \quad T_n}{\varphi} \quad (\Delta)$$

where φ is a formula for $\mathcal{S}$, Δ is an inference rule in $\mathcal{L}$, and $T_1,\ldots,T_n$ are proof trees such that

$$\frac{head(T_1) \quad \cdots \quad head(T_n)}{\varphi}$$

is an instance of Δ for the theory $\mathcal{S}$.

We say that a proof tree is *closed* whenever it is finite and contains no open goals. [6]

Notice the difference between φ, an open goal, and $\overline{\varphi}$, a goal closed by a rule without premises.

Definition 3.2 A proof tree T is a *proper prefix* of a proof tree T' if there are one or more open goals $\varphi_1,\ldots,\varphi_n$ in T such that T' is obtained from T by replacing each φ_i by a non-atomic proof tree T_i having φ_i as its head. We denote this as $T \subset T'$.

An *infinite proof tree* is an infinite increasing chain of finite trees, that is, a sequence $\{T_i\}_{i\in\mathbb{N}}$ such that for all i, $T_i \subset T_{i+1}$.

We characterize the proof trees with computational meaning (those which are computed by an *interpreter* [25]), by means of the notion of well-formed proof tree.

Definition 3.3 We say that a proof tree T is *well-formed* if it is either an open goal, or a closed proof tree, or a proof tree of the form

$$\frac{T_1 \quad \cdots \quad T_n}{\varphi} \quad (\Delta)$$

where, for each j, T_j is itself well-formed, and there is $i \leq n$ such that T_i is not closed, for any $j < i$, T_j is closed, and each of the T_{i+1} ,...,T_n is an open goal. An infinite proof tree is *well-formed* if it is an ascending chain of well-formed finite proof trees. $\mathcal{S}$ is called *operationally terminating* if no infinite well-formed tree for $\mathcal{S}$ exists.

So operational termination intuitively means that, given an initial goal, an interpreter that solves goals from left to right will either succeed in finite time in producing a closed proof tree, or will fail in finite time, not being able to close or extend further any of the possible proof trees, after exhaustively searching all such proof trees.

[6] Open goals appear at the leaves of a proof tree; but they can be *closed* by the application of inference rules with no premises. For example, an open goal $t \to t$ can be closed by applying a Reflexivity inference rule.

4 A transformational approach to termination of programs

In this paper we study the termination problem for rewrite theories, and informally describe a number of theory transformations Θ which have been developed so far and that can be composed in various ways. These transformations are nontermination preserving (or *termination reflecting*), i.e., given a theory $\mathcal{R}$ in a given logic $\mathcal{L}$, the operational termination of $\Theta(\mathcal{R})$ in a given logic $\mathcal{L}'$ implies the operational termination of $\mathcal{R}$ w.r.t. $\mathcal{L}$. Thus, they can in the end map a rewrite theory to a transformed TRS that can be proved terminating with standard tools.

Before being able to describe these transformations, we briefly sketch the different kind of logics/theories/programs that we transform here. Due to lack of space, we cannot provide full technical details, but we provide the appropriate references to more precise descriptions.

4.1 Rewrite theories (RWT)

A rewriting logic specification is called a *rewrite theory* (RWT) [4]. It is a tuple $\mathcal{R} = (\Sigma, E \cup Ax, \mu, R, \phi)$, where:

- $(\Sigma, E \cup Ax)$ is a membership equational (MEL) theory: Σ is an order-sorted signature [16], Ax is a set of (equational) axioms, and E is a set of sentences

$$t = t' \quad \text{if} \quad A_1, \ldots, A_n \qquad \text{or} \qquad t : s \quad \text{if} \quad A_1, \ldots, A_n$$

  where the A_i are atomic equations or memberships $t_i : s_i$ establishing that term t_i has sort s_i [3,29]. Since we are often interested in distinguishing the *MEL* component within a rewrite theory, we refer to it as $\mathcal{R}_T$, i.e., $\mathcal{R}_T = (\Sigma, E \cup Ax)$ for $\mathcal{R}$ as above. Furthermore, we often (shortly) denote a rewrite theory $\mathcal{R}$ as $\mathcal{R} = (\mathcal{R}_T, \mu, R, \phi)$ when the underlying MEL theory $\mathcal{R}_T$ is clear from the context.
- $\mu : \Sigma \to \mathcal{P}_{fin}(\mathbb{N})$ is a mapping specifing for each $f \in \Sigma$ the argument positions under which subterms can be simplified with the equations in E [21,22].
- R is a set of *labeled conditional rewrite rules* of the general form

$$r : (\forall X)\ q \longrightarrow q' \ \text{ if } \ (\bigwedge_i u_i = u_i') \wedge (\bigwedge_j v_j : s_j) \wedge (\bigwedge_l w_l \longrightarrow w_l').$$

- $\phi : \Sigma \to \mathcal{P}_{fin}(\mathbb{N})$ is a mapping assigning to each function symbol $f \in \Sigma$ (with, say, n arguments) a set $\phi(f) \subseteq \{1, \ldots, n\}$ of *frozen positions* under which it is forbidden to perform any rewrites with rules in R.

Intuitively, $\mathcal{R}$ specifies a *concurrent system*, whose states are elements of the initial algebra $T_{\Sigma/E \cup Ax}$ and whose *concurrent transitions* are specified by the rules R, subject to the frozenness constraints imposed by ϕ. Therefore, mathematically each state is modeled as an $(E \cup Ax)$-equivalence class $[t]_{E \cup Ax}$ of ground terms, and rewriting happens *modulo* $E \cup Ax$, that is, R rewrites not just terms t but rather $(E \cup Ax)$-equivalence classes $[t]_{E \cup Ax}$ representing states.

$$(R\text{-Reflexivity}) \qquad \frac{t \to_E^* t'}{t \to_R^* t'}$$

$$(R\text{-Transitivity}) \qquad \frac{t \to_R^1 t' \qquad t' \to_R^* t''}{t \to_R^* t''}$$

$$(R\text{-Congruence}) \qquad \frac{u_i \to_R^1 u_i'}{f(u_1, \ldots, u_i, \ldots u_n) \to_R^1 f(u_1, \ldots, u_i', \ldots, u_n)} \quad \text{where } i \notin \phi(f)$$

$$(R\text{-Replacement}) \qquad \frac{u \to_E^* u' \;\; A_1^\bullet\sigma \;\; \ldots \;\; A_n^\bullet\sigma \;\; t'\sigma \to_E^* v}{u \to_R^1 v}$$

where $t \to t'$ if $A_1 \cdots A_n$ in $R^\bullet$ and $u' =_{Ax} t\sigma$

Fig. 1. Inference rules for executing rewrite theories

The execution semantics is defined by the inference system in Figure 1, which uses the inference system of Figure 2, as an auxiliary subsystem and involves the two rewriting relations $\to_E$ and $\to_R$ (in both one-step and reflexive-transitive variants), as well as the ':' and '::' membership relations. Here, $t :: s$ is a subrelation of the relation $t : s$, corresponding to the special case of a membership in which the term t is not further rewritten with $\to_E$ before computing its sort (see [11]). To distinguish between $\to_E$ and $\to_R$ we adopt the convention of decorating all rewrite relations in the subinference system of Figure 2 with E. So they now appear as either $\to_E^1$ or $\to_E^*$ in that subsystem.

4.2 Sugared Membership Rewrite Theories (SCS-MCTRSs)

By a *sugared* context-sensitive membership rewrite theory (SCS-MCTRS) we understand a tuple $\mathcal{R} = (\Sigma, S, \leq, \mu, Ax, R, M)$ where [26]:

(i) S is a set of *sorts* and $(S, \leq)$ is a partial order.

(ii) $\Sigma = \Sigma_0 \uplus \Sigma_1$, where Σ_0 contains the symbols which are given an explicit sort in the SCS-MCTRS specification, whereas Σ_1 contains symbols that do not admit a profile based only on 'proper' sorts but rather require the use of *kinds* (corresponding to the connected components in $(S, \leq)$ as a whole [7]). Such use of kinds is typically needed for functions that are *intrinsically partial*. For example, given a sort `Path` of paths in a graph, a binary path concatenation function has to be declared at the kind level as `_;_ : [Path] [Path] -> [Path]`, because it is intrinsically partial on pairs of paths: it is undefined unless the target node of the first path coincides with the source node of the

[7] The connected components of $(S, \leq)$ can be thought of as the equivalence classes $S/\equiv_\leq$, where $\equiv_\leq$ is the smallest equivalence relation containing the order $\leq$.

(Subject reduction) $$\frac{t \to^1 t' \qquad t' : s}{t : s}$$

(Membership-1) $$\frac{A_1^\bullet\sigma \quad \cdots \quad A_n^\bullet\sigma}{u :: s}$$
where $t : s$ if $A_1 \cdots A_n$ in R_T and $u =_{Ax} t\sigma$

(Membership-2) $$\frac{t :: s}{t : s}$$

(Reflexivity) $$\frac{}{t \to^* t'} \quad \text{if } t =_{Ax} t'$$

(Transitivity) $$\frac{t \to^1 t' \qquad t' \to^* t''}{t \to^* t''}$$

(Congruence) $$\frac{u_i \to^1 u_i'}{f(u_1, \ldots, u_i, \ldots u_n) \to^1 f(u_1, \ldots, u_i', \ldots, u_n)}$$
where $i \in \mu(f)$

(Replacement) $$\frac{A_1^\bullet\sigma \quad \ldots \quad A_n^\bullet\sigma}{u \to^1 t'\sigma}$$
where $t \to t'$ if $A_1 \cdots A_n$ in R_T and $u =_{Ax} t\sigma$

Fig. 2. Inference rules for membership rewrite theories

second path.

(iii) As for rewrite theories, $\mu : \Sigma \to \mathcal{P}_{fin}(\mathbb{N})$ is a mapping sending each symbol f accepting n arguments to a subset $\mu(f) \subseteq \{1, \ldots, n\}$.

(iv) Ax is a collection of axioms such as associativity, commutativity.

(v) R is a set of *conditional rewrite rules* of the form

$$(\forall X)\, t \to t' \text{ if } A_1 \wedge \ldots \wedge A_k$$

where the A_i are either rewrite conditions $u \to v$, or memberships $w : s$.

(vi) M is a set of *conditional memberships* of the form

$$(\forall X)\, t : s \text{ if } A_1 \wedge \ldots \wedge A_k$$

with the A_i as before.

The inference system in Figure 2 defines the execution semantics of SCS-MCTRSs.

4.3 Conditional Term Rewriting Systems and Context-Sensitivity

We refer the reader to [32] to recall the usual notions and notations regarding term rewriting and CTRSs. In general, a conditional rewrite rule is as follows:

$$l \to r \;\text{ if }\; s_1 = t_1, \cdots, s_n = t_n$$

where $l, r, s_1, t_1, \cdots, s_n, t_n$ are terms (without any sort or kind information and discipline). Terms l and r are called the left- and right-hand sides of the rule, and the sequence $s_1 = t_1, \cdots, s_n = t_n$ (often denoted c) is the *conditional part* of the rule. We are mainly concerned with *oriented* CTRSs whose (conditional) rules are written as follows:

$$l \to r \;\text{ if }\; s_1 \to t_1, \cdots, s_n \to t_n$$

indicating that the conditions $s_i \to t_i$ for $1 \leq i \leq n$ are intended to express the *reachability*, in arbitrarily *many steps*, of (instances of) t_i from (instances of) s_i.

We also consider two further generalizations of the CTRS notion. First, we want to allow rewriting *modulo* a set Ax of equational axioms, so that matching of rules is performed with an Ax-matching algorithm. We therefore view such a CTRS as a triple $\mathcal{R} = (\Sigma, Ax, R)$ with Σ the signature of function symbols, Ax the equational axioms we rewrite modulo, and R the set of conditional rewrite rules. A second generalization is making rewriting *context-sensitive* [21,22] so that only certain function arguments are rewritten, whereas other arguments remain "frozen". For example, it is natural to restrict the evaluation of an **if-then-else** operator so that rewriting is only allowed on the first argument. In this way, we can express that the evaluation of the conditions only makes sense after evaluating the guard of the conditional expression. The simplest way of specifying requirements of this kind is to assume that there is a *replacement map* [21], i.e., a function $\mu : \Sigma \longrightarrow \mathcal{P}(\mathbf{N})$ associating to each operator f of n arguments a set of argument positions $\mu(f) = \{i_1, \ldots, i_m\}$, with $1 \leq i_j \leq n$, which are those under which rewriting is allowed. For example, $\mu(\textbf{if-then-else}) = \{1\}$, and in Example 1.1 $\mu(\text{cons}) = \{1\}$. A context-sensitive CTRS (CS-CTRS) is a pair $(\mathcal{R}, \mu)$, with $\mathcal{R}$ a CTRS that may involve axioms Ax and a replacement map μ.

4.4 Sketch of the transformations

The overall family of composable nontermination-preserving transformations is summarized in Figure 3. In the following sections, we briefly describe how these transformations proceed and which is the main focus for each of them.

5 From SRWTs to SCS-MCTRSs: merging equations and rules (transformation C)

Perhaps the simplest theory transformation we can attempt in order to reduce the operational termination of an SRWT $\mathcal{R} = (\Sigma, E \cup Ax, \mu, R, \phi) = (\mathcal{R}_T, \mu, R, \phi)$ to a simpler termination problem is to merge equations E and rules R (transformation

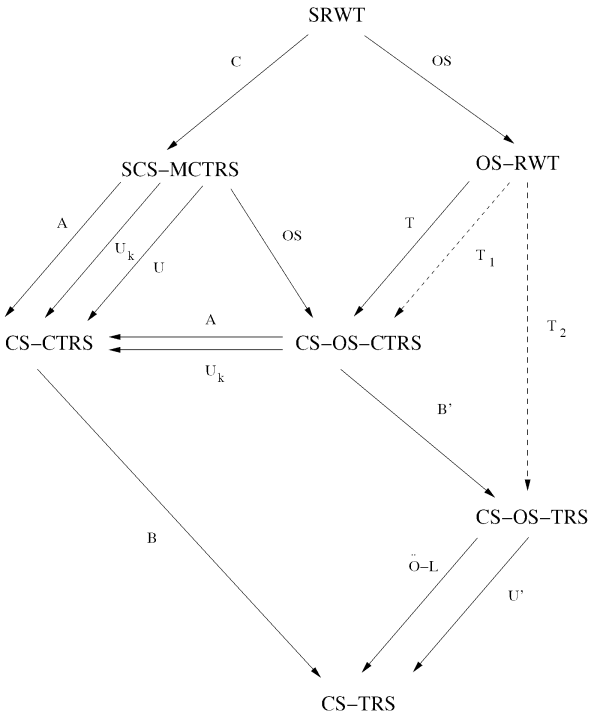

Fig. 3. Transformations for proving termination of Rewrite Theories

C [9]). This can be achieved under the assumption that μ and ϕ are *complementary* maps, that is, for any function symbol f with n arguments, and for any i, $1 \leq i \leq n$, we have $i \in \mu(f)$ if and only if $i \notin \phi(f)$.

The theory transformation $\mathcal{R} \mapsto C(\mathcal{R})$ transforms $\mathcal{R}$ into the (S)CS-MCTRS $C(\mathcal{R})$. This transformation reduces the problem of proving the operational termination of $\mathcal{R}$ (under the inference system of Figure 1, plus the auxiliary inference subsystem of Figure 2) to proving the operational termination of the (S)CS-MCTRS $C(\mathcal{R})$ under the simpler inference system of Figure 2. The transformation extends $(\mathcal{R}_T, \mu)$ by just adding a new sort `Truth` to the set of sorts, a new constant `tt` of that sort, and a new operator `equal` of sort `Truth` to the signature, and by further adding to $\mathcal{R}_T$ rules $\texttt{equal}(x : [s], x : [s]) \to tt$ for each kind $[s]$, and the following set R° of rules:

$$R^\circ = \{t \to t' \text{ if } A_1^\circ, \ldots, A_n^\circ \mid (t \to t' \text{ if } A_1, \ldots, A_n) \in R\}$$

where if A_i is a membership then $A_i^\circ = A_i$, if A_i is a matching equation $u_i = v_i$, then A_i° is the rewrite condition $v_i \to^* u_i$, if A_i is an ordinary equation $u_i = v_i$, then A_i° is the rewrite condition $\texttt{equal}(u_i, v_i) \to^* \texttt{tt}$, and if A_i is a rewrite condition $w_i \to q_i$, then A_i° is the rewrite condition $w_i \to^* q_i$. That is, we wipe out any distinction between E and R in the conditions of R (note that $\mathcal{R}_T$ never contained such distinctions).

Example 5.1 The rewrite theory expressed as a system module in Example 1.4 becomes a *functional* module (the `equal` operator is not needed):

```
fmod MARKS-LISTS-C is
  sorts Nat List MNat MList .
  subsort List < MList .
  subsort Nat < MNat .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op # : -> MNat .
  op nil : -> List .
  op _;_ : Nat List -> List .
  op _;_ : MNat MList -> MList .
  op <_> : MList -> MList .
  vars M N N1 N2 N3 : Nat .
  vars L L' : List .
  vars X : MNat .
  vars XS : MList .
  ceq < L > = < # ; L > if < L > = < N1 ; N2 ; N3 ; L' > .
  eq  # ; (N ; M ; L) = N ; (# ; M ; L) .
  eq  # ; N ; L = L .
endfm
```

There is no distinction now between equations, matching conditions and rules.

6 From SCS-MCTRSs/CS-OS-CTRSs to CS-CTRSs: encoding sort information (transformation A)

The transformation A [11] allows us to deal with *sort information* (subsort declarations, rank declarations for symbols in the signature, sorted variables occurring in equations or rules,...) of an SCS-MCTRSs or a CS-OS-CTRSs.

We add a truth-value constant tt, plus unary operators is_s, is'_s for each $s \in S$. Here, predicates is_s deal with sort declarations for variables like $x : s$ where x is a variable and *no reduction* below is_s in an instance of $is_s(x)$ is required to check the membership (hence we further let $\mu(is_s) = \emptyset$). On the other hand, predicates is'_s are intended to deal with 'proper' memberships $w : s$, where w is a nonvariable term (or s is a *membership sort*). In order to appropriately check such memberships, the obtained sort expressions $is'_s(w)$ may require some *subject reduction*; thus we let $\mu(is'_s) = \{1\}$ to enable such reductions. We have new rules $is'_s(x) \to is_s(x)$ for each sort $s \in S$. In this way, we implement the idea that $\_:: s$ (represented by predicates is_s) is a subrelation of $\_: s$ (represented by predicates is'_s): if $is_s(t)$ holds (i.e., it rewrites to tt), then $is'_s(t)$ also holds. Each conditional rule $t \to t'$ if $A_1, \ldots, A_n$ involving variables $x_1 : s_1, \ldots, x_m : s_m$; becomes a conditional rule of the form,

$$t \to t' \text{ if } \{is_{s_i}(x_i) \to \mathrm{tt}\}_{1 \le i \le m},\ \widetilde{A}_1, \ldots, \widetilde{A}_n \tag{1}$$

where if A_i is a membership $u_i : s'_i$, then: (i) if u_i is a nonvariable term, then $\widetilde{A}_i$ is the rewrite condition $is'_{s'_i}(\widetilde{u_i}) \to \mathrm{tt}$, and (ii) if $u_i \equiv x$ is a variable, then $\widetilde{A}_i$ is the rewrite condition $is_{s'_i}(x) \to \mathrm{tt}$; otherwise, if A_i is a rewrite condition $u_i \to v_i$, then $\widetilde{A}_i$ is the rewrite condition $\widetilde{u_i} \to \widetilde{v_i}$. Finally, we replace each conditional membership $t : s$ if $A_1, \ldots, A_n$ involving variables $x_1 : s_1, \ldots x_m : s_m$, by a conditional rule

$$is_s(\widetilde{t}) \to \mathrm{tt} \text{ if } \{is_{s_i}(x_i) \to \mathrm{tt}\}_{1 \le i \le m},\ \widetilde{A}_1, \ldots, \widetilde{A}_n. \tag{2}$$

In this way, type checking within a membership condition $t : s$ (corresponding to the sorted variables $x_1 : s_1, \ldots, x_m : s_m$ occurring in t) is handled by predicates is_{s_i}, $1 \le i \le m$.

Example 6.1 The CS-CTRS obtained from the SCS-MCTRS in Example 1.1 is:

```
fmod LengthOfFiniteListsAndTake-A is
   sort S .
   op isKNat : S -> S [strat (0)] .            *** Kind predicates
   op isKNatIList : S -> S [strat (0)] .
   op isNat : S -> S [strat (0)] .             *** Sort predicates: 'primed' versions are not
   op isNatIList : S -> S [strat (0)] .        *** necessary due to the absence of 'proper'
   op isNatList : S -> S [strat (0)] .         *** membership
   op tt : -> S .
   op and : S S -> S .
   op 0 : -> S .                               *** The unsorted signature begins
   op s : S -> S .
   op zeros : -> S .
   op nil : -> S .
   op cons : S S -> S [strat (1 0)] .
   op take : S S -> S .
   op length : S -> S .                        *** End of the unsorted signature

   vars T M N IL L : S .                       *** Unsorted variables

   eq isKNat(0) = tt .                         *** Definition of kind predicates
   ceq isKNat(s(N)) = tt if isKNat(N) = tt .
   ceq isKNat(length(L)) = tt if isKNatIList(L) = tt .
   eq isKNatIList(nil) = tt .
   eq isKNatIList(zeros) = tt .
   ceq isKNatIList(cons(N,IL)) = tt if isKNat(N) = tt /\ isKNatIList(IL) = tt .
   ceq isKNatIList(take(N,IL)) = tt if isKNat(N) = tt /\ isKNatIList(IL) = tt .

   ceq isNatIList(IL) = tt if isNatList(IL) = tt .  *** Implementation of subsorting

   eq isNat(0) = tt .                          *** Sorting for the symbols in the signature
   ceq isNat(s(N)) = tt if isNat(N) = tt .
   ceq isNat(length(L)) = tt if isNatList(L) = tt .
   eq isNatIList(zeros) = tt .
   ceq isNatIList(cons(N,IL)) = tt if isNat(N) = tt /\ isNatIList(IL) = tt .
   eq isNatList(nil) = tt .
   ceq isNatList(cons(N,L)) = tt if isNat(N) = tt /\ isNatList(L) = tt .
   ceq isNatList(take(N,IL)) = tt if isNat(N) = tt /\ isNatIList(IL) = tt .

   eq zeros = cons(0,zeros) .                  *** Transformed rules begin
   ceq take(0,IL) = nil if isKNatIList(IL) = tt /\ isNatIList(IL) = tt .
   ceq take(s(M),cons(N,IL)) = cons(N,take(M,IL)) if isKNat(M) = tt /\ isKNat(N) = tt /\
         isKNatIList(IL) = tt /\ isNat(M) = tt /\ isNat(N) = tt /\ isNatIList(IL) = tt .
   ceq length(nil) = 0 .
   ceq length(cons(N,L)) = s(length(L)) if isKNat(N) = tt /\ isKNatList(L) = tt /\
         isNat(N) = tt /\ isNatList(L) = tt .
endfm
```

Transformations U_K and U were also discussed in [11] as increasingly simpler lightweight variants of A: U_K ignores kind information, but still encodes sort information as predicates; whereas U ignores both kind ans sort information.

7 From SCS-MCTRSs to CS-OS-CTRSs: dealing with explicit memberships (transformation OS)

The transformation OS, mapping an SCS-MCTRS to a CS-OS-CTRS, is described in detail in [26]. An SCS-MCTRS does already have an order-sorted signature, with a poset of sorts $(S, \leq)$. The corresponding order-sorted signature for the transformed CS-OS-CTRS has a new top sort for each connected component in $(S, \leq)$. Furthermore, we add a new sort, *Truth*, unrelated to all previous sorts, with a constant *tt*. However, we must *remove* from this signature all so-called *membership sorts* (see [26]), which intuitively correspond to sorts where non-sugared

memberships may be intrinsically needed to determine whether a term has that sort. All other sorts are called *order-sorted sorts*. While membership of a term in an order-sorted sort can be determined syntactically by the exclusive use of an order-sorted parsing algorithm, membership of a term in a membership sort cannot be so determined; it is instead axiomatized in the transformed theory by adding to its signature new *Truth*-valued predicates for each membership sort that return *tt* when applied to a term in the transformed theory if and only if that term has that sort in the original theory.

Example 7.1 The `PALINDROME` program above can be viewed as an SCS-MCTRS. After applying transformation OS, we obtain the following CS-OS-CTRS [8]:

```
fmod PALINDROME-OS is
  sorts Qid List Pal [List] [Truth] .
  subsorts Qid < Pal < List < [List] .
  op tt : -> [Truth] .
  op nil : -> [Pal] .
  op __ : [List]  [List]  -> [List]  [assoc id: nil] .
  op is'-Qid : [List] -> [Truth].
  op is'-Pal : [List] -> [Truth].
  op is'-List : [List] -> [Truth].
  op is-Pal : [List] -> [Truth] [strat (0)] .
  op is-List : [List] -> [Truth] [strat (0)] .
  var I : Qid .
  var P : [Pal] .
  var K : [List] .
  vars L L' : List .
  ceq is-Pal(I P I) = tt if is-Pal(P) = tt .
  eq is'-Pal(K) = is-Pal(K) .
  eq is'-List(K) = is-List(K) .
  eq is'-Qid(I) = tt .
  eq is'-Pal(I) = tt .
  eq is-Pal(I) = tt .
  eq is'-List(I) = tt .
  eq is-List(I) = tt .
  ceq is-List(L L') = tt if is-List(L) = tt /\ is-List(L') = tt .
  ceq is-List(K) = tt if is-Pal(K) = tt .
endfm
```

In contrast, the SCS-MCTRS `LengthOfFiniteListsAndTake` remains *unchanged* under transformation OS!

8 From SRWTs to OS-RWT: dealing with explicit memberships in rewrite theories (transformation OS)

A very important transformation maps a SRWT $\mathcal{R}$ to a corresponding OS-RWT $OS(\mathcal{R})$. This is just a slight generalization of the transformation from a SCS-MCTRS to a CS-OS-CTRS in Section 7, which is extended in a straightforward way to our desired transformation $\mathcal{R} \mapsto OS(\mathcal{R})$. The corresponding transformation $\mathcal{R} \mapsto OS(\mathcal{R})$ has now a very simple description. If $\mathcal{R} = (\mathcal{R}_T, \mu, R, \phi)$, then $OS(\mathcal{R}) = (OS(\mathcal{R}_T, \mu), OS(R), OS(\phi))$, where $(\mathcal{R}_T, \mu) \mapsto OS(\mathcal{R}_T, \mu)$ is the just-summarized transformation from a SCS-MCTRS to a CS-OS-CTRS, $OS(R)$ contains for each rule $t \to t'$ if $A_1, \ldots, A_n$ in R a corresponding rule with the same left-

[8] Note that we use brackets for giving names to *sorts* in the obtained OS-CS-CTRS; despite this 'kind-like' notation, no kinds are actually present here!

and right-hand sides, but where: (i) all variables having a membership sort remain unchanged; and all variables having a kind have been replaced by variables of the corresponding new top sort for the connected component of sorts for that kind; (ii) all variables x having a membership sort s have been replaced by variables of the corresponding new top sort for the connected component of that membership sort, and an additional condition of the form $is_s(x) \longrightarrow^*_E tt$; (iii) any condition A_i of the form $w : s$ for some sort s is replaced by a condition of the form $is'_s(w) \longrightarrow^*_E tt$, (where, as explained in [26] and in Section 6, the difference between the is_s and is'_s predicates is that is'_s allows equational reduction of its argument, whereas is_s does not); and (iv) all other conditions A_j are left unchanged. Finally, the frozenness mapping $OS(\phi)$ extends the original ϕ in a straightforward way by agreeing with ϕ on the old function symbols and considering all arguments of all new function symbols added to the signature as unfrozen. The end result is that the transformed theory $OS(\mathcal{R})$ is an OS-RWT, as desired.

Example 8.1 The program `MARKS-LISTS` remains *unchanged* under transformation `OS`.

9 From OS-RWTs to CS-OS-CTRSs: encoding equational rewriting (transformation T)

Given an order-sorted rewrite theory $\mathcal{R} = (\Sigma, S, \leq, E \cup Ax, \mu, R, \phi)$, we define a transformation $\mathcal{R} \mapsto T(\mathcal{R})$, where $T(\mathcal{R}) = (\Sigma', S', \leq', Ax', E' \cup R', \mu')$ is an OS-CS-CTRS, and therefore has a single rewrite relation. Here:

- $S' \supset S$ extends S by adding a fresh new sort *True*, and for each connected component C of sorts (which need not have a top sort), a fresh new sort C'; and $\leq'$ extends $\leq$ only by the identity relations $C' \leq' C'$, and *True* $\leq'$ *True*.
- $\Sigma' \supset \Sigma$ extends Σ by adding: (i) a constant tt of sort *True*; (ii) for each connected component of sorts C an operator $eq : C'\, C' \longrightarrow$ *True*; and (iii) for each connected componet C of sorts and each maximal sort $s \in C$ two new operators:

$$[\_], \{\_\} : s \longrightarrow C'$$

- μ' extends μ by the declarations $\mu'([\_]) = \emptyset$, $\mu'(\{\_\}) = \emptyset$, and $\mu'(eq) = \emptyset$.
- $Ax' \supset Ax$ extends Ax by declaring each eq commutative.
- E' consists of the following rules:
  · For each (possibly conditional) equation

$$t = t' \;\; \textit{if} \;\; A_1, \ldots, A_n \tag{3}$$

  in E, rules

$$t \to t' \;\; \textit{if} \;\; A^\bullet_1, \ldots, A^\bullet_n \tag{4}$$

$$\{t\} \to [t'] \;\; \textit{if} \;\; A^\bullet_1, \ldots, A^\bullet_n \tag{5}$$

  where: if A_i is a matching equation $u_i = v_i$, then $A^\bullet_i$ is the rewrite condition

$[v_i] \to [u_i]$, and if A_i is an ordinary equation $u_i = v_i$, then $A_i^\bullet$ is the rewrite condition $eq([u_i],[v_i]) \to tt$.

- The following rules are given for eq (for s, s' (not necessarily distinct) maximal sorts in the same connected component, with x, z of sort s, and y of sort s'):

$$eq([x],[x]) \longrightarrow tt \tag{6}$$

$$eq([x],[y]) \longrightarrow eq([z],[y]) \;\; \textit{if} \;\; \{x\} \longrightarrow [z] \tag{7}$$

- For each nonconstant f in Σ having a maximal arity $s_1 \ldots s_n$ and each i in $\mu(f)$ we add a rule (with x_j of sort s_j, and y of sort s_i)

$$\{f(x_1,\ldots,x_i,\ldots,x_n)\} \longrightarrow [f(x_1,\ldots,y,\ldots,x_n)] \;\; \textit{if} \;\; \{x_i\} \to [y] \tag{8}$$

- for each maximal sort s in the subsort ordering of $(S, \leq)$, with variables x, y of sort s we add the rule

$$[x] \longrightarrow [y] \;\; \textit{if} \;\; \{x\} \to [y] \tag{9}$$

- For each rule $t \longrightarrow t' \;\; \textit{if} \;\; A_1,\ldots,A_n$ in R, we get in R' the rule $t \longrightarrow t' \;\; \textit{if} \;\; A_1^\bullet,\ldots,A_n^\bullet$ where $A_i^\bullet$ is defined as above; plus the case of conditions of the form $u \longrightarrow v$, which are left without change.

Example 9.1 The program `MARKS-LISTS-OS` (which coincides with `MARKS-LISTS`, see Example 8.1) is transformed by T as follows:

```
mod MARKS-LISTS-OS-T is
  sorts List MList MNat Nat Thruth [MList] [MNat] .
  subsort List < MList .
  subsort Nat < MNat .
  op # : -> MNat .
  op 0 : -> Nat .
  op <_> : MList -> MList .
  op _;_ : MNat MList -> MList .
  op _;_ : Nat List -> List .
  op [_] : MList -> [MList] [frozen (1)] .
  op [_] : MNat -> [MNat] [frozen (1)] .
  op {_} : MList -> [MList] [frozen (1)] .
  op {_} : MNat -> [MNat] [frozen (1)] .
  op equal : [MList] [MList] -> Thruth [frozen (1 2)] .
  op equal : [MNat] [MNat] -> Thruth [frozen (1 2)] .
  op nil : -> List .
  op s : Nat -> Nat .
  op tt : -> Thruth .
  crl [introduce] : < L:List > => < # ; L:List >
    if [< L:List >] => [< N1:Nat ; N2:Nat ; N3:Nat ; L:List >] .
  rl [propagate] : # ; N:Nat ; M:Nat ; L:List => N:Nat ; # ; M:Nat ; L:List .
  rl [remove] : # ; N:Nat ; L:List => L:List .
  rl equal([X:MList], [X:MList]) => tt .
  rl equal([X:MNat], [X:MNat]) => tt .
  crl {< X1:MList >} => [< Y:MList >] if {X1:MList} => [Y:MList] .
  crl {X1:MNat ; X2:MList} => [X1:MNat ; Y:MList] if {X2:MList} => [Y:MList] .
  crl {X1:MNat ; X2:MList} => [Y:MNat ; X2:MList] if {X1:MNat} => [Y:MNat] .
  crl {X1:Nat ; X2:List} => [X1:Nat ; Y:List] if {X2:List} => [Y:List] .
  crl {X1:Nat ; X2:List} => [Y:Nat ; X2:List] if {X1:Nat} => [Y:Nat] .
  crl {s(X1:Nat)} => [s(Y:Nat)] if {X1:Nat} => [Y:Nat] .
endm
```

Note that if the theory $\mathcal{R}$, besides satisfying conditions (1)–(3), is such that: (i) the equations E are unconditional; and (ii) in any rule $t \longrightarrow t' \;\; \textit{if} \;\; A_1,\ldots,A_n$ in R, all the conditions A_i are non-equational rewrite conditions, then the above transformation $\mathcal{R} \mapsto T(\mathcal{R})$ can be greatly simplified: we do not need the new sorts and the new operators tt, eq, $[\_]$, and $\{\_\}$, so that the signature remains unchanged. And we do not need to add any extra, auxiliary rules at all: we just convert the equations E into rules, and leave the rules R unchanged. We denote by T_1 this

simpler transformation. An even simpler case is when, in addition, (iii) the rules R are unconditional. Then we just turn the equations into rules and try to prove the termination of the OS-CS-TRS with unconditional rules $E \cup R$ modulo A. We then denote the transformation by T_2. It is just exactly like T_1, but it has the advantage that $T_2(\mathcal{R})$ is always an *unconditonal* OS-TRS.

Yet a different kind of simplification can be obtained when $E = \emptyset$ but R has equational conditions. If such equational conditions include ordinary equations, then we just need to add tt, eq, and $[\_]$, and just rules of the form $eq(x, x) \longrightarrow tt$. Furthermore, if all equational conditions only involve matching equations, then we can also ignore tt and eq, and only need to add $[\_]$.

10 Final transformations to a CS-TRS

The transformations sketched in Sections 5 to 9 show how to deal with the features of rewriting logic programs. They finally yield (possibly together with some underlying set of axioms) either a context-sensitive, order-sorted conditional rewrite system (CS-OS-CTRS) or a context-sensitive, conditional rewrite system (CS-CTRS). Despite the fact that no termination tool deals with such kind of systems directly, it is possible to further transform them into a context-sensitive term rewriting system (CS-TRS) for which we can obtain an automatic proof of termination by using tools like AProVE or MU-TERM. Transformation B from CS-CTRS to CS-TRSs (described in [11]) generalizes to the CS-case a well-known transformation form CTRSs to TRSs described, e.g., in [32]. Transformation B' from CS-OS-CTRS to CS-OS-TRSs (described in [26]) plays a similar role for the order-sorted case. The transformation Ö-L from CS-OS-TRS to CS-TRSs (described in [26]) generalizes to the CS level a well-known transformation by Ölveczky and Lysne [33].

Thus, given a rewrite theory $\mathcal{R}$, which we assume in sugared form (SRWT), we can always transform it, in a way that preserves operational nontermination, into a CS-TRS, which can then be sent to a number of automatic termination tools, so that a proof of termination of this transformed CS-TRS yields a proof of operational termination for our original rewrite theory.

11 Conclusions and further work

We have studied the problem of proving the operational termination of rewrite theories having expressive features such as the distinction between equations E and rules R, sorts, subsorts, membership predicates, rewriting modulo axioms, and context-sensitive rewriting for both equations and rules. Our approach is transformational and relies on the preservation of operational nontermination in the transformations we propose. We have implemented all these transformations in the MAUDE Termination Tool (MTT, http://www.lcc.uma.es/~duran/MTT). Our initial experiments suggest that these transformations can be effective in proving termination of a wide range of rewriting logic programs. However, we believe that the techniques presented here should be combined with more *intrinsic* techniques,

for example to keep sort and subsort information around and to use it directly in termination proofs rather than encoding such sort information into conditions. For instance, the following specification of the factorial function [27]:

```
fmod FACTORIAL is
  sorts Nat NzNat .
  subsorts NzNat < Nat .
  op 0 : -> Nat .
  op s : Nat -> NzNat .
  op p : NzNat -> Nat .
  op _+_ : Nat Nat -> Nat .
  op _+_ : NzNat Nat -> NzNat .
  op _+_ : NzNat NzNat -> NzNat .
  op _*_ : Nat Nat -> Nat .
  op _*_ : NzNat NzNat -> NzNat .
  op fact : Nat -> NzNat .
  vars x y : Nat .
  vars x' : NzNat .
  eq x + 0 = x .
  eq x + s(y) = s(x + y) .
  eq x * 0 = 0 .
  eq x * s(y) = x + (x * y) .
  eq fact(0) = s(0) .
  eq fact(x') = x' * fact(p(x')) .
  eq p(s(x)) = x .
endfm
```

can be easily proved terminating (as an Order-Sorted Term Rewriting System) by using the recently introduced order-sorted dependency pairs method [27], implemented as part of the tool MU-TERM. In contrast, we could not obtain an automatic proof of termination using the transformations described above. Thus, developing direct methods for proving termination of programs at the different theory levels depicted in Figure 3 is an interesting subject for future work.

References

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

[2] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.

[3] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Comput. Sci.*, 236:35–132, 2000.

[4] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science* 351(1):386-414, 2006.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. All About Maude – A High-Performance Logical Framework. Lecture Notes in Computer Science 4350, 2007.

[6] CoFI Task Group on Semantics. CASL—The common algebraic specification language, version 1.0, Semantics. http://www.brics.dk/Projects/CoFI/Documents/CASL/Semantics/index.html, 1999.

[7] E. Contejean and C. Marché, B. Monate and X. Urbain. Proving termination of rewriting with C*i*ME. In *Proc. of WST'03*, pages 71-73, Technical Report DSIC II/15/03, Valencia, Spain, 2003. Available at http://cime.lri.fr.

[8] F. Durán, S. Lucas, C. Marché, J. Meseguer, and X. Urbain. Proving Termination of Membership Equational Programs. In P. Sestoft and N. Heintze, editors, *Proc. of ACM SIGPLAN 2004 Symposium PEPM'04*, pages 147–158. ACM Press, 2004.

[9] F. Durán, S. Lucas, and J. Meseguer. Operational Termination in Rewriting Logic. Technical Report 2008. http://www.dsic.upv.es/~slucas/tr08.pdf

[10] F. Durán, S. Lucas, and J. Meseguer. MTT: The Maude Termination Tool. In *Proc. of IJCAR'08*, LNCS 5195:313-319, Springer-Verlag, Berlin, 2008.

[11] F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving Operational Termination of Membership Equational Programs. *Higher-Order and Symbolic Computation*, 21(1-2):59–88, 2008.

[12] S. Eker. Term Rewriting with Operator Evaluation Strategies. In C. Kirchner and H. Kirchner, editors, *Proc. of 2nd International Workshop on Rewriting Logic and its Applications, WRLA'98*, Electronic Notes in Computer Science, 15(1998):1-20, 1998.

[13] K. Futatsugi and R. Diaconescu. *CafeOBJ Report*. World Scientific, AMAST Series, 1998.

[14] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Proc. of IJCAR'06*, LNAI 4130:281-286, Springer-Verlag, Berlin, 2006.

[15] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In *Proc of RTA'06*, LNCS 4098:297-312, Springer Verlag, Berlin, 2006.

[16] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[17] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*, Kluwer, 2000.

[18] N. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Computation*, 205:474-511, 2007.

[19] P. Hudak, S. Peyton-Jones, and P. Wadler. Report on the Functional Programming Language Haskell: a non–strict, purely functional language. *SIGPLAN Notices*, 27:1–164, 1992.

[20] A. Koprowski. TPA: Termination Proved Automatically. In *Proc of RTA'06*, LNCS 4098:257-266, Springer Verlag, Berlin, 2006. http://www.win.tue.nl/tpa

[21] S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1), 1-61, 1998.

[22] S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):294–343, 2002.

[23] S. Lucas. MU-TERM: A Tool for Proving Termination of Context-Sensitive Rewriting In *Proc. of 15h RTA'04*, LNCS 3091:200-209, Springer-Verlag, Berlin, 2004. Available at http://www.dsic.upv.es/~slucas/csr/termination/muterm.

[24] S. Lucas. Termination of on-demand rewriting and termination of OBJ programs. In *Proc. of 3rd International Conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 82-93, ACM Press, 2001.

[25] S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95:446–453, 2005.

[26] S. Lucas and J. Meseguer. Operational Termination of Membership Equational Programs: the Order-Sorted Way. In *Proc. of WRLA'08*, *Electronic Notes in Theoretical Computer Science*, to appear, 2009.

[27] S. Lucas and J. Meseguer. Order-Sorted Dependency Pairs. In *Proc. of 10th International Conference on Principles and Practice of Declarative Programming, PPDP'08*, pages 108-119, *ACM Press*, 2008.

[28] J. Meseguer. General logics. In *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[29] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proceedings WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.

[30] J. Meseguer and J. Goguen. Initiality, induction and computability. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge University Press, 1985.

[31] U. Nilsson and J. Maluszynski. Logic, Programming and Prolog (2ed) John Wiley & Sons, 1995

[32] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, Berlin, 2002.

[33] P.C. Ölveczky and O. Lysne. Order-Sorted Termination: The Unsorted Way. In *Proc. of ALP'96*, LNCS 1139:92-106, Springer-Verlag, Berlin, 1996.

[34] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Analysis for Logic Programs by Term Rewriting. In *Proc. of LOPSTR'06 (selected papers)*, LNCS 4407:177-193, Springer-Verlag, Berlin, 2007.

[35] TeReSe, editor, *Term Rewriting Systems*, Cambridge University Press, 2003.