Proving Termination of Membership Equational Programs *

Francisco Durán
LCC, Universidad de Málaga, Spain

Salvador Lucas
DSIC, Universidad Politécnica de Valencia, Spain

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

Claude Marché
PCRI, LRI (CNRS UMR 8623), INRIA Futurs, Université Paris-Sud, France

Xavier Urbain
PCRI, LRI (CNRS UMR 8623), INRIA Futurs, Université Paris-Sud, France

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

General Terms

Languages, Theory, Verification

Keywords

Membership Equational Logic, Term Rewriting, Termination

ABSTRACT

Advanced typing, matching, and evaluation strategy features, as well as very general conditional rules, are routinely used in equational programming languages such as, for example, ASF+SDF, OBJ, CafeOBJ, Maude, and equational subsets of ELAN and CASL. Proving termination of equational programs having such expressive features is important but nontrivial, because some of those features may not be supported by standard termination methods and tools, such as mu-term, C*i*ME, AProVE, TTT, Termptation, etc. Yet, use of the features may be essential to ensure termination. We present a sequence of theory transformations that can be used to bridge the gap between expressive equational programs and termination tools, prove the correctness of such transformations, and discuss a prototype tool performing the transformations on Maude equational programs and sending the resulting transformed theories to some of the aforementioned tools.

*This research was partly supported by bilateral CNRS-DSTIC/UIUC research project "Rewriting calculi, logic and behavior", and by ONR Grant N00014-02-1-0715 and NSF Grant CCR-0234524; Francisco Durán and Salvador Lucas were partially supported by Spanish MCYT grant STREAM TIC2001-2705-C03-01.

PEPM'04, August 24–26, 2004, Verona, Italy.


```
fmod OvConsOS 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
```

Figure 1: Example of Maude program

1. INTRODUCTION

The goal of this work is to study transformational techniques that can help to bridge the gap between programs in expressive rule-based equational languages such as ASF+SDF [30], OBJ [15], Maude [8], CafeOBJ [12], and modules in suitable equational subsets of ELAN [3] and CASL [2] on the one hand, and termination tools assuming considerably more restrictive specifications (untyped, unconditional, etc.). There is a clear tension between the goals of expressiveness and efficiency when using equational theories as *programs*, and the considerably simpler assumptions of standard reasoning techniques for rewrite systems and their associated tools. For example, many equational programs do not terminate in the usual sense, but do so when evaluated with suitable *types* and/or *strategies*.

Example 1. *Consider the Maude specification in Figure 1, 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 both for building finite and infinite lists of natural numbers and is declared with evaluation* strategy $(1\ 0)$. *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 in the annotation. Note also that* `NatList` *is a subsort of* `NatIList`*, thus allowing the use of* `take` *to extract items both from finite and infinite lists.*

This system is terminating, but both the evaluation strategy $(1\ 0)$ *for* `cons` *and the use of sorts (especially* `NatList` *and* `NatIList` *instead of a single one) are crucial to achieve this terminating behavior. In fact, by either removing the strategy annotation or the sort information we would get a non-terminating program.*

Some termination tools are not able to deal directly with such programs because they make use of either types or strategies, or because of other features such as conditional equations that are not handled by a given tool's input language. As remarked in Example 1, forgetting such extra features is often insufficient in order to prove termination, because termination may crucially depend on the extra features being erased. Expressive features not handled by some current termination tools include:

1. Sorts, subsorts, overloading and membership predicates;
2. Conditions, which may introduce extra variables;
3. Fixed evaluation strategies (e.g., leftmost innermost or leftmost outermost);
4. Programmable evaluation strategies which permit annotating each function symbol with local strategy information on what arguments to evaluate or not (e.g., context sensitive rewriting strategies [19], E-strategies [15, 8], etc.);
5. Rewriting modulo axioms like associativity (A), commutativity (C), identity (I), AC, ACI, and so on.

For example, AProVE [14] supports some form of conditional equations (2) and innermost rewriting (3), but none of the other features; C*i*ME [9] directly supports part of (5), but not (1)-(4); whereas MU-TERM [21] directly supports (4) but not the rest.

Our goal is to leverage a wide range of termination tools, including those just mentioned, by using a sequence of *theory transformations* that map the original program into increasingly simpler theories —each having the property that termination of the transformed theory at each step ensures termination of the input theory— until we reach a transformed theory that we can enter into a tool. We first provide a new theoretical framework which allows us to deal with very expressive programs having all the features (1)–(5) mentioned above, so as to make our techniques applicable to as many equational programming languages as possible. Then, we transform it by a sequence of transformation steps eliminating, successively, features (1), (2) and (5). In this paper we just ignore (3), because indeed innermost rewriting with a conditional TRS is not clearly defined, see Section 5 for further discussion.

The endpoint of this transformation process is a *CS-TRS*, i.e., a TRS (Term Rewriting System) together with a *replacement map* μ which discriminates, for each symbol of the signature, the argument positions $\mu(f)$ at which replacements are allowed (thus reflecting item (4) above). The notion of rewriting which deals with such replacement restrictions is called *context-sensitive rewriting* (*CSR* [18, 19]). Some research effort has already been devoted to the definition and implementation of techniques for proving termination of *CSR* (see, e.g., [4, 10, 13, 17, 22, 32]).

The sequence of theory transformations is summarized in Figure 2. Transformation A eliminates memberships and sorts (feature 1) resulting in an *unsorted*, *context-sensitive* and *conditional* rewrite theory. Transformation B eliminates conditions, possibly with extra variables (feature 2) in a way that generalizes a known transformation from Conditional Term Rewriting Systems (CTRS) to TRS [25] by making it aware of context-sensitive rewriting information; in this way we obtain an *unsorted* and *unconditional context-sensitive* rewrite theory. At this point, two options are available, leading to the forking in Figure 2. On the one hand, we can use a termination tool (such as MU-TERM) that can directly prove termination of *CSR* as explained in, e.g., [4, 22] (left branch). On the other hand, we can use several existing theory transformations, including those proposed by Lucas [17], Zantema [32], Ferreira and Ribeiro [10], and Giesl and Middeldorp [13], to pass from a context-sensitive rewrite theory to an ordinary rewrite theory whose termination ensures that of the context-sensitive theory. These transformations are also implemented in MU-TERM. The resulting theory can then be sent to a number of termination tools (e.g., C*i*ME, AProVE, TERMPTATION, TTT, etc.). Our overall goal is of course to bridge the gap between expressive equational programs with features (1)–(5), and termination tools *all the way through*.

This paper is organized as follows: in Section 2, we recall basics of Membership Equational Logic, Membership Rewrite Theories, and their operational semantics. In Section 3 we describe our theory transformations and prove their soundness w.r.t termination: transformation A is defined in Section 3.1, transformation B in Section 3.2. The example in Figure 1 is used as a running example for these transformations. In Section 4, we discuss implementation issues, and give more examples.

2. REWRITING WITH MEMBERSHIP EQUATIONAL THEORIES

2.1 Membership Equational Theories

Since membership equational theories generalize both many-sorted and order-sorted equational theories and can also deal with partiality [24], they are quite expressive from the typing point of view and can therefore provide a quite general type theory for equational programs. We can describe a membership signature as a triple, $\Omega = (K, \Sigma, S)$, where (K, Σ) is a K-sorted signature, that is, K is a set, and Σ is an indexed family of sets $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ —that we call "many-kinded" because the elements of K are called *kinds* so as to avoid confusion with the sorts S that are instead treated as predicates—and $S = \bigcup_{k\in K} S_k$ is a disjoint family of unary predicates. Each $s \in S_k$ is called a *sort*, and is understood as a unary predicate on k, written $\_ : s$, so that elements satisfying the predicate determine the extension of the sort s in k. Intuitively, elements having some sort s are well-defined elements, whereas elements having a kind k but no sort are understood as error elements. A model of Ω, called a membership algebra B is a (K, Σ)-algebra B together with an interpretation for each unary predicate $s \in S_k$, whose extension is a subset $B_s \subseteq B_k$.

Ω-sentences are then universally quantified Horn clauses whose atomic predicates are either equalities $t = t'$ between two Σ-terms of the same kind, or unary membership predicates $t : s$ with t a Σ-term of kind k and $s \in S_k$. In other words, membership equational logic is just the sublogic of many-sorted (although we see it here as "many-kinded") Horn clause logic with equality in which all the predicates other than equality are unary. A membership equational *theory* is just a pair $T = (\Omega, E)$ with E a set of

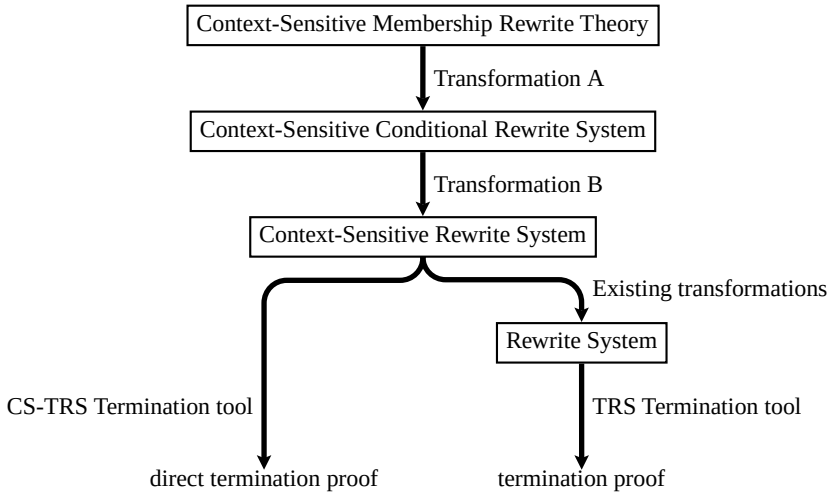

Figure 2: Overview of the methodology

Ω-sentences. T-algebras are then Ω-algebras satisfying the equations E in T, according to the usual notion of satisfaction in many-sorted (again, seen as "many-kinded") first-order logic with equality. Given a membership equational theory T, there are free and initial T-algebras, and sound and complete inference rules [24]. Order-sorted notation $s_1 < s_2$ for subsorts can be used to abbreviate the conditional membership $(\forall x : k)\ x : s_2$ if $x : s_1$. Similarly, an operator declaration $f\colon s_1 \times \cdots \times s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom $(\forall x_1 : k_1, \ldots, x_n : k_n)\ f(x_1, \ldots, x_n) : s$ if $\bigwedge_{1 \leq i \leq n} x_i : s_i$. We write $(\forall x_1 : s_1, \ldots, x_n : s_n)\ t = t'$ in place of $(\forall x_1 : k_1, \ldots, x_n : k_n)\ t = t'$ if $\bigwedge_{1 \leq i \leq n} x_i : s_i$. The above abbreviations make it easy to embed order-sorted specifications as a special case of the more general membership equational specifications. Specifically, an *order-sorted specification* is one in which: (1) the only memberships are subsort declarations $s_1 < s_2$ and operator declarations $f\colon s_1 \times \cdots \times s_n \to s$; and (2) the only other clauses in E are conditional equations of the form $(\forall x_1 : s_1, \ldots, x_n : s_n)\ t = t'$ if $\bigwedge_{1 \leq i \leq n} u_i = v_i$. The Maude language [8] supports all the order-sorted abbreviations just mentioned; furthermore, kinds do not have to be declared explicitly by the user: they are inferred by the system, that associates a kind to each connected component of sorts in the subsort ordering graph. For example, the specification in Figure 1 is order-sorted and has two kinds, corresponding to the connected components {`Nat`} and {`NatList`, `NatIList`}.

Admissible membership equational theories [8] provide a very general class of equational theories that are executable by equational rewriting. Their sentences are a union $E \cup Ax$, where Ax is a collection of equational axioms such as, for example, associativity, commutativity, and identity of some operators in Σ, for which a matching algorithm modulo Ax exists, and E consists of *conditional equations* (1) and conditional *memberships* (2):

$$t = t' \quad \text{if} \quad A_1, \ldots, A_n \tag{1}$$

$$t : s \quad \text{if} \quad A_1, \ldots, A_n \tag{2}$$

where in (1) the variables in t' are among those in t or in some A_i, and where, in both (1) and (2) each A_i is either a membership $w_i : s_i$, or an equation $u_i = v_i$ such that any new variable not in t or in some A_j with $j < i$ must occur only in u_i or in some A_j with $j > i$; furthermore, if u_i introduces any new variables, then u_i must be a nonvariable term; we then call $u_i = v_i$ a *matching equation*. In MAUDE such matching equations are distinguished syntactically with the notation $u_i := v_i$.

2.2 From Membership Equational Theories to Membership Rewrite Theories

In the spirit of [6], we can associate to an admissible membership equational theory $T = (\Omega, E \cup Ax)$ a corresponding conditional rewrite theory R_T defined as follows. The signature of R_T adds a fresh new kind Truth with a constant tt, and for each kind k in T an operator $\mathrm{equal} : k\ k \longrightarrow \mathrm{Truth}$. R_T has the same equational axioms Ax as T, so that rewriting is performed modulo Ax, and contains rules of the form $\mathrm{equal}(x, x) \to \mathrm{tt}$ for each kind k in T. Furthermore, for each admissible conditional equation of the form (1) in E there is a conditional rule of the form

$$t \to t' \quad \text{if} \quad \hat{A}_1, \ldots, \hat{A}_n \tag{3}$$

where if A_i is a membership then $\hat{A}_i = A_i$, if A_i is a matching equation $u_i = v_i$, then $\hat{A}_i$ is the rewrite condition $v_i \to u_i$, and if A_i is an ordinary equation $u_i = v_i$, then $\hat{A}_i$ is the rewrite condition $\mathrm{equal}(u_i, v_i) \to \mathrm{tt}$. Similarly, for each conditional membership in T of the form (2) we associate a conditional membership of the form,

$$t : s \quad \text{if} \quad \hat{A}_1, \ldots, \hat{A}_n \tag{4}$$

with the $\hat{A}_i$ defined exactly as before.

2.3 Computing with Context-Sensitive Membership Rewrite Theories

The point of associating to an admissible membership equational theory T a corresponding rewrite theory R_T is that we can perform equational reasoning by rewriting. Of course, unless R_T satisfies additional properties such as confluence, sort-decreasingness, and so on, equational reasoning by rewriting will only be sound but not necessarily complete.

Equational reasoning in an equational theory T by rewriting with the rules in R_T modulo the axioms Ax can be made more expressive by requiring 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. As done in *context-sensitive rewriting* (*CSR* [18, 19]) the simplest way of specifying requirements of this kind is to assume that there is a *replacement map* [18], i.e., a function $\mu : \Sigma \longrightarrow \mathcal{P}(\mathbf{N})$ associating to each operator f of n arguments a set of argument positions $\mu(f) = \{i_1, \ldots, i_k\}$, with $1 \leq i_j \leq n$, which are those under which rewriting is allowed. For example, μ(if-then-else) $= \{1\}$, and in Example 1 μ(cons) $= \{1\}$. An important application of *CSR* is that rewrite systems that are nonterminating if rewriting is allowed on all term positions can often become terminating. As we show below for the example in Figure 1 (two other examples are discussed in Section 4), this allows us to handle infinite data structures. We then call the pair (R_T, μ) a *context-sensitive membership rewrite theory* (CS-MRT), where the context information is provided by μ. For instance, the CS-MRT specification (also given in Maude-like notation) which corresponds to the Maude program in Figure 1 is given in Figure 3. Here, `[Nat]` denotes the kind

```
fmod OvConsOSMRT is
   kind [Truth] .
   kind [Nat].
   kind [NatIList] .
   op tt : -> [Truth] .
   op 0 : -> [Nat] .
   op s : [Nat] -> [Nat] .
   op zeros : -> [NatIList] .
   op nil : -> [NatList] .
   op cons : [Nat] [NatIList] -> [NatIList] [strat (1 0)] .
   op take : [Nat] [NatIList] -> [NatIList] .
   op length : [NatIList] -> [Nat] .
   cmb L : NatIList if L : NatList .
   mb tt : Truth .
   mb 0 : Nat .
   cmb s(N) : Nat if N : Nat .
   mb zeros : NatIList .
   mb nil : NatList .
   cmb cons(N,IL) : NatIList if N : Nat /\ IL : NatIList .
   cmb cons(N,L) : NatList if N : Nat /\ L : NatList .
   cmb take(N,IL) : NatList if N : Nat /\ IL : NatIList .
   cmb length(L) : Nat if L : NatList .
   eq zeros = cons(0,zeros) .
   ceq take(0,IL) = nil if IL : NatIList .
   ceq take(s(M),cons(N,IL)) = cons(N,take(M,IL))
      if M : Nat /\ N : Nat /\ IL : NatIList .
   eq length(nil) = 0 .
   ceq length(cons(N,L)) = s(length(L))
      if N : Nat /\ L : NatList .
endfm
```

Figure 3: CS-MRT for the program OvConsOS

of sort `Nat`, and `[NatIList]` denotes the kind of both sorts `NatList` and `NatIList`. The profile of the operators is given in terms of these kinds. We omit the operator `equal` as no conditional rule includes equations in its conditional part. Note also the first conditional membership (with keyword `cmb`) which expresses that `NatList` is a subsort of `NatIList`. The sort profile for the arguments and result of each operator in the Maude program `OvConsOS` are desugared here as memberships in the CS-MRT specification. In particular, viewing the sort profile of a function symbol as a shorthand for a kind profile together with a membership, such as for `cons` above, allows us to cleanly handle operator overloading: to each different sort profile corresponds a different membership. We also allow *ad-hoc* overloading, that is, operators with same name and different kind profile, although in that case we require that if f has kind profiles $k_1 \cdots k_n \rightarrow k$ and $k_1 \cdots k_n \rightarrow k'$, then $k = k'$.

We can define the rewriting relation associated to (R_T, μ) by means of the inference rules of Figure 4, which adapt to the context-sensitive case those in Figure 7 in [6]. Note that all the inferences are implicitly assumed to happen *modulo* the equational axioms Ax in R_T. The new relation $t \rightarrow t' : s$ combines rewriting and sort inference; intuitively it means that t can be rewritten to a term t' for which we can infer the sort s. For each atom B appearing in a condition of a conditional rule or a conditional membership in R_T we use the meta-notation $B^\bullet$ to denote: (1) if B is a rewrite $u \longrightarrow v$ (including the case when $u = eq(w, w')$ and $v = tt$), then $B^\bullet = B$; (2) if B is of the form $x : s$ with x a variable, then $B^\bullet = x \longrightarrow x : s$; and if B is of the form $w : s$ with w a nonvariable term, then $B^\bullet = w \longrightarrow x : s$, with x a fresh new varible of the kind of s. The inference system in Figure 4 is context sensitive in a quite detailed way. The most obvious case is the Congruence rule, which blocks rewriting in argument positions outside $\mu(f)$; further context sensitivity is achieved through the $B^\bullet$ conjuncts in the conditions of the Membership and Replacement rules. The point is that, if unrestricted, these inference rules could easily undermine context-sensitivity by evaluating subterms that are supposed to be frozen, thus easily leading to nontermination. This is prevented by the case when $B = x : s$, since then $B^\bullet = x \longrightarrow x : s$. This means that if x matches a subterm of the term whose sort we are computing with the Membership rule —or that we are trying to rewrite with the Replacement rule— then that subterm will not be further rewritten in the process of checking its sort..

Given a context-sensitive membership rewrite theory (R_T, μ), we write $(R_T, \mu) \vdash t \rightarrow t' : s$ and $(R_T, \mu) \vdash t \rightarrow^* u$ whenever $t \rightarrow t' : s$, resp. $t \rightarrow u$, are derivable using the rules of Figure 4 (note that because of reflexivity and transitivity, $t \rightarrow u$ may involve zero, one, or more rewrite steps). The relation $\rightarrow^*$ is by definition the reduction relation in 0, 1 or more steps for (R_T, μ). We write $(R_T, \mu) \vdash t \rightarrow^+ u$ whenever $t \rightarrow u$ is derivable using the rules above, with at least one application of (Replacement) which does not appear just in proofs of premises of (Subject reduction) or (Membership) (that is, not just as a subproof of a relation $t' \rightarrow t'' : s$) but contributes directly to the proof of $t \rightarrow u$. In the special case of a standard rewrite theory without any memberships, without context-sensitive restrictions, and with only unconditional rules, the definition above for $\rightarrow^*$ (resp. $\rightarrow^+$) corresponds to the usual definition of the reflexive-transitive (resp. transitive) closure of the one step rewrite relation. As in [6], it is easy to prove by translating each rewriting step into an equational inference step the following soundness theorem.

THEOREM 1 (SOUNDNESS). *Let T be such that all kinds are nonempty. Then, $(R_T, \mu) \vdash t \rightarrow^* u$ implies $T \vdash t = u$. Similarly, $(R_T, \mu) \vdash t \rightarrow t' : s$ implies $T \vdash t : s$, $T \vdash t' : s$, and $T \vdash t = t'$*
.

DEFINITION 1. *We say that a membership equational program is terminating whenever the relation $\rightarrow^+$ is well-founded.*

3. THEORY TRANSFORMATIONS

3.1 Transformation A: From Membership Theories to Unsorted Rewrite Theories

Let T be an admissible membership rewrite theory $((K, \Sigma, S), E \cup Ax)$, we define an *unsorted* conditional rewrite theory $\widetilde{T} = (\widetilde{\Sigma}, \widetilde{Ax}, R(K, \Sigma, S) \cup \widetilde{E})$ as follows. $\widetilde{\Sigma} = \{\widetilde{\Sigma}_i\}_{i \geq 0}$ consists of the constant tt, the binary operator equal, and for

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

(Membership) $$\frac{A_1^\bullet\sigma \quad \cdots \quad A_n^\bullet\sigma}{t\sigma \to t\sigma : s} \qquad \text{where} \quad t : s \ \text{ if } \ A_1 \cdots A_n \quad \text{ in } R_T$$

(Reflexivity) $$\frac{}{t \to t}$$

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

(Congruence) $$\frac{u_{i_1} \to u'_{i_1} \quad \cdots \quad u_{i_k} \to u'_{i_k}}{f(u_1, \ldots, u_n) \to f(u_1, \ldots, u'_{i_1}, \ldots, u'_{i_k}, \ldots, u_n)} \qquad \text{where } \mu(f) = \{i_1, \ldots, i_k\}$$

(Replacement) $$\frac{A_1^\bullet\sigma \quad \ldots \quad A_n^\bullet\sigma}{t\sigma \to t'\sigma} \qquad \text{where} \quad t \to t' \ \text{ if } \ A_1 \cdots A_n \quad \text{ in } R_T$$

Figure 4: Inference rules for membership rewriting

each operator $f : w \longrightarrow k$ where $w = k_1 \ldots k_n$ in Σ, a new operator name $f^w \in \widetilde{\Sigma}_n$. We furthermore add unary operators $is_k \in \widetilde{\Sigma}_1$ for each $k \in K$, and $is_s \in \widetilde{\Sigma}_1$ for each $s \in S$. The role of operators f^w is to disambiguate ad-hoc overloading: for each Σ-term t, the $\widetilde{\Sigma}$-term $\widetilde{t}$ is obtained by making its variables unsorted, and by replacing each $f : w \to k$ by f^w. As said at end of Section 2.1, there should be only one k for each w, so this operation is well-defined. The axioms $\widetilde{Ax}$ are just the equations $\widetilde{t} = \widetilde{t'}$ for each $t = t'$ in Ax. The rules in $R(K, \Sigma, S)$ include the rule $\mathrm{equal}(x, x) \to \mathrm{tt}$, plus conditional rules of the form

$$is_k(f^w(x_1, \ldots, x_n)) \to \mathrm{tt} \quad \text{ if } \quad \{is_{k_i}(x_i) \to \mathrm{tt}\}_{1 \le i \le n} \quad (5)$$

for each $f : w \longrightarrow k$ in Σ, with $w = k_1 \ldots k_n$. The set of conditional rules $\widetilde{E}$ contains for each conditional rule of the form (3) and involving variables $x_1 : k_1, \ldots, x_m : k_m$ a conditional rule of the form,

$$\widetilde{t} \to \widetilde{t'} \quad \text{ if } \quad \{is_{k_i}(x_i) \to \mathrm{tt}\}_{1 \le i \le m},\ \widetilde{A}_1, \ldots, \widetilde{A}_n \quad (6)$$

where if $\hat{A}_i$ is a membership $u_i : s_i$, then $\widetilde{A}_i$ is the rewrite condition $is_{s_i}(\widetilde{u_i}) \to \mathrm{tt}$; and if $\hat{A}_i$ is a rewrite condition $u_i \to v_i$, then $\widetilde{A}_i$ is the rewrite condition $\widetilde{u_i} \to \widetilde{v_i}$, and leaving the symbol equal unchanged. Likewise, $\widetilde{E}$ contains for each conditional membership of the form (4) and involving variables $x_1 : k_1, \ldots x_m : k_m$ a conditional rule for the form,

$$is_s(\widetilde{t}) \to \mathrm{tt} \ \text{ if } \ \{is_{k_i}(x_i) \to \mathrm{tt}\}_{1 \le i \le m},\ \widetilde{A}_1, \ldots, \widetilde{A}_n \quad (7)$$

The above translation extends in a straightforward way to a translation $(R_T, \mu) \mapsto (\widetilde{T}, \widetilde{\mu})$ of context-sensitive rewrite theories, where $\widetilde{\mu}(f^w) = \mu(f)$ for each $f : w \longrightarrow k$ in Σ, $\widetilde{\mu}(\mathrm{equal}) = \{1, 2\}$, and for each $k \in K$ and each $s \in S$ we define $\widetilde{\mu}(is_k) = \emptyset$ and $\widetilde{\mu}(is_s) = \emptyset$ (this choice avoids unexpected reductions during the kind- and sort-checking phases implemented by these functions).

EXAMPLE 2. *For our running example, we would get the transformed system in Figure 5. We have omitted the disambiguation of operators, since no ambiguity is involved in this example; also,* equal *has been omitted.*

3.1.1 Preservation of termination

We first have the following lemma, proved by straightforward structural induction, making use of rules of type (5).

LEMMA 1. *For any term t, substitution σ, condition c, $\widetilde{t\sigma} = \widetilde{t}\,\widetilde{\sigma}$ and $\widetilde{(c\sigma)} = \widetilde{c}\,\widetilde{\sigma}$. If t is a ground term, well-kinded of kind k w.r.t T, then $(\widetilde{T}, \widetilde{\mu}) \vdash is_k(\widetilde{t}) \to^+ \mathrm{tt}$.*

LEMMA 2. *For any ground terms t and u, and any sort s:*

- *if $(R_T, \mu) \vdash t \to u : s$ then $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{t} \to \widetilde{u}$ and $(\widetilde{T}, \widetilde{\mu}) \vdash is_s(\widetilde{u}) \to^+ \mathrm{tt}$, and*
- *if $(R_T, \mu) \vdash t \to^* u$ then $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{t} \to^* \widetilde{u}$*

Moreover if $(R_T, \mu) \vdash t \to^+ u$ then $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{t} \to^+ \widetilde{u}$.

PROOF. We prove both assertions simultaneously, by induction on the proof trees of $(R_T, \mu) \vdash t \to u : s$ and $(R_T, \mu) \vdash t \to^* u$ respectively. If $(R_T, \mu) \vdash t \to u : s$, it is derived either

- By rule (Subject reduction): we have $(R_T, \mu) \vdash t \to^* u$ and $(R_T, \mu) \vdash u \to^* u : s$. By induction, we have $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{t} \to^* \widetilde{u}$ and $(\widetilde{T}, \widetilde{\mu}) \vdash is_s(\widetilde{u}) \to^+ \mathrm{tt}$, hence $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{t} \to^* \widetilde{u}$.
- By rule (Membership): there is a membership rule $l : s \ \text{ if } \ c$ in R_T and a substitution σ such that $t = l\sigma$, and $(R_T, \mu) \vdash c\sigma$. By induction, we have $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{c\sigma} = \widetilde{c}\,\widetilde{\sigma}$. In $\widetilde{T}$ there is the rule $is_s(\widetilde{l}) \to \mathrm{tt} \ \text{ if } \ \{is_{k_i}(x_i) \to \mathrm{tt}\}_{1 \le i \le m},\ \widetilde{c}$. To apply (Replacement) with substitution $\widetilde{\sigma}$ to get $(\widetilde{T}, \widetilde{\mu}) \vdash is_s(\widetilde{t}) \to \mathrm{tt}$, we need to show that the is_{k_i} conditions are satisfied, that is for each i, $is_{k_i}(x_i\widetilde{\sigma}) \to \mathrm{tt}$: this is a consequence of Lemma 1.

If $(R_T, \mu) \vdash t \to^* u$, it was derived either

- By rule (Reflexivity) that is $t = u$, hence $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{t} \to^* \widetilde{u}$ also by (Reflexivity).
- By rule (Transitivity): there is a term w such that $(R_T, \mu) \vdash t \to^* w$ and $(R_T, \mu) \vdash w \to^* u$. By induction, we have $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{t} \to^* \widetilde{w}$ and $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{w} \to^* \widetilde{u}$, hence $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{t} \to^* \widetilde{u}$ again by (Transitivity).
- By rule (Congruence) that is $t = f(t_1, \ldots, t_k)$, $u = f(u_1, \ldots, u_k)$, $\mu(f) = \{i_1 \ldots i_k\}$ and for each j, $(R_T, \mu) \vdash t_{i_j} \to^* u_{i_j}$. By induction, $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{t_{i_j}} \to^* \widetilde{u_{i_j}}$ and since in the new strategy map $\widetilde{\mu}(f_s) = \mu(f)$, we conclude again by (Congruence).

```
fmod OvConsOSMRT_TA is
   sort S .
   op isKNat : S -> S [strat (0)] .
   op isKNatIList : S -> S [strat (0)] .
   op isNat : S -> S [strat (0)] .
   op isNatIList : S -> S [strat (0)] .
   op isNatList : S -> S [strat (0)] .
   op tt : -> S .
   op and : S S -> S .
   op 0 : -> S .
   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 .
   vars T M N IL L : S .
   eq isKNat(0) = tt .
   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 .
   eq isNat(0) = tt .
   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) .
   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
```

Figure 5: Use of transformation A

- by rule (Replacement): there exists a rule $l \to r$ if c in R_T and a substitution σ such that $t = l\sigma$, $u = r\sigma$, and $(R_T, \mu) \vdash c\sigma$. By induction, $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{c\sigma} = \widetilde{c}\,\widetilde{\sigma}$. In $\widetilde{T}$, there is the rule $\widetilde{l} \to \widetilde{r}$ if $\{is_{k_i}(x_i) \to \mathrm{tt}\}_{1 \le i \le m}, \widetilde{c}$. We again conclude by (Replacement), as in the (Membership) case above.

If $(R_T, \mu) \vdash t \to^+ u$, then it is straightforward to see that $(\widetilde{T}, \widetilde{\mu}) \vdash \widetilde{t} \to^+ \widetilde{u}$ by examining each of the four cases above. □

THEOREM 2. *If the system $(\widetilde{T}, \widetilde{\mu})$ resulting from the transformation of (R_T, μ) is terminating, then (R_T, μ) is also terminating.*

PROOF. Any infinite reduction for (R_T, μ) could be lifted into an infinite reduction for $(\widetilde{T}, \widetilde{\mu})$, using the above lemma. □

3.1.2 Optimizations

In order to provide the simplest input for the next transformation which removes conditions from rules (see Section 3.2), we can apply some obvious optimizations on the previous transformation which do not change the termination behavior of the program.

1. In a first variant, the is_k predicates for kinds are omitted; this simplifies the resulting theory with minimal loss in its expressiveness, particularly for specifications in which, as it is usually the case, all variables of a conditional equation or rule are required to have a sort in the condition.

2. If all operator profiles involve only sorts, and all variables appearing in equations and memberships have a declared sort, then if k is the kind of a sort s, then $is_s(x) \to tt$ implies $is_k(x) \to tt$. Therefore, we can safely use $is_s(x) \to tt$ instead of $is_k(x) \to tt \wedge is_s(x) \to tt$ in the conditional part of the rules computed by the transformation.

3. A conditional part like

   $$is_{s_1}(x_1) \to tt \wedge \cdots \wedge is_{s_k}(x_k) \to tt$$

   in a conditional rule can be collapsed into a single expression

   $$and(is_{s_1}(x_1), and(\cdots, is_{s_k}(x_k)) \cdots) \to tt$$

   at the end of the conditional part by introducing a binary '`and`' operator defined by

   ```
   op and : S S -> S .
   eq and(tt,T) = T .
   ```

   Moreover, if the right hand side of the conditional rule is tt, we can use the previous expression with `and` as the new right hand-side of the rule: the conditional rule

   $$l \to tt \text{ if } is_{s_1}(x_1) \to tt \wedge \cdots \wedge is_{s_k}(x_k) \to tt$$

   eventually collapses into the unconditional one

   $$l \to and(is_{s_1}(x_1), and(\cdots, is_{s_k}(x_k)) \cdots)$$

For instance, after implementing these optimizations, the equations of the previous system become as shown in Figure 6. Note that many conditional rules become unconditional now, and that the conditional parts of the remaining conditional rules have been greatly compressed.

On the other hand, there is a much simpler variant of the transformation $(R_T, \mu) \mapsto (\widetilde{T}, \tilde{\mu})$ just defined. For *order-sorted* rewrite theories, in which the only memberships involved in conditions are variables, and the only membership axioms correspond to subsort

and operator declarations, a second variant drops also the is_s predicates for sorts. Obviously, since these simpler variants yield less restrictive conditions in the translated rules in $\widetilde{T}$, these variants allow more rewrites and therefore our results apply *a fortiori* to these simpler transformations, in the sense that a termination proof for the transformed theory ensures termination of the original theory. These variants afford simpler transformations at the cost of looser simulations of the rewrites. For instance, it is *not* possible to use this variant to prove termination of program `OvConsOS` in Figure 1. in fact, the obtained CS-TRS:

with $\mu(\texttt{cons}) = \{1\}$ and $\mu(f) = \{1, \ldots, ar(f)\}$ for all other symbols f, is non-terminating:

$$\begin{array}{rcl} \texttt{length(}\underline{\texttt{zeros}}\texttt{)} & \hookrightarrow & \underline{\texttt{length(cons(0,zeros))}} \\ & \hookrightarrow & \texttt{s(length(}\underline{\texttt{zeros}}\texttt{))} \\ & \hookrightarrow & \cdots \end{array}$$

where $\hookrightarrow$ means context-sensitive rewriting, i.e., rewriting restricted to those positions on which reductions are allowed by the replacement map ([18, 19]) Here, the information about sorts is essential for ensuring termination. In fact, reducing `length(cons(0,zeros))` into `s(length(zeros))` is not possible in `OvConsOS` because `zeros` does not belong to the sort `NatList`. In Section 4, however, we give an interesting and successful example of the use of this simpler transformation variant as part of our transformation process.

As related work, we want to mention the property of persistence introduced by Zantema [31]: a property P of a many-sorted TRS R is *persistent* if P is true on R iff it is true on $T(R)$, where T is the simpler, sort-removing transformation just mentioned above. Zantema proved that termination is persistent for restricted classes of TRS. Note, however, that we are using a different notion of termination, namely termination of *CSR*. Persistence of termination of *CSR* has not been studied yet. On the other hand, we do not need persistence: we only need preservation of termination, i.e., we need that if $T(R)$ terminates, then R terminates and do not need the opposite to hold.

3.2 Transformation B: from Unsorted Conditional to Unconditional

The next step is to define a transformation associating to an admissible and unsorted conditional and context-sensitive rewrite theory (T, μ) an unconditional one (T^*, μ^*). We extend Ohlebusch's transformation [25] so as to handle the context-sensitive restrictions imposed by the replacement map μ.

```
eq isNatIList(IL) = isNatList(IL) .
eq isNat(0) = tt .
eq isNat(s(N)) = isNat(N) .
eq isNat(length(L)) = isNatList(L) .
eq isNatIList(zeros) = tt .
eq isNatIList(cons(N,IL)) = and(isNat(N),isNatIList(IL)) .
eq isNatList(nil) = tt .
eq isNatList(cons(N,L)) = and(isNat(N),isNatList(L)) .
eq isNatList(take(N,IL)) = and(isNat(N),isNatIList(IL)) .
eq zeros = cons(0,zeros) .
ceq take(0,IL) = nil
    if isNatIList(IL) = tt .
ceq take(s(M),cons(N,IL)) = cons(N,take(M,IL))
    if and(isNat(M),and(isNat(N),isNatIList(IL)))))) = tt .
eq length(nil) = 0 .
ceq length(cons(N,L)) = s(length(L))
    if and(isNat(N),isNatList(L)))) = tt .
```

Figure 6: Optimized transformation A

```
zeros → cons(0,zeros)
take(0,IL) → nil
take(s(M),cons(N,IL)) → cons(N,take(M,IL))
length(nil) → 0
length(cons(N,L)) → s(length(L))
```

Let $T = (\Sigma, Ax, R)$ be an unsorted conditional rewrite theory, then $T^* = (\Sigma^*, Ax, R^*)$ where Σ^* extends Σ, and μ^* extends μ by introducing a set of new operators U_i as defined below, and R^* is the set of rules obtained from R as follows: for each conditional rule $l \to r$ if $a_1 \to b_1, \ldots, a_n \to b_n$ we introduce $n+1$ unconditional rules

$$\begin{array}{rcll} l & \to & U_1(a_1, \vec{x_1}) & \\ U_{i-1}(b_{i-1}, \vec{x_{i-1}}) & \to & U_i(a_i, \vec{x_i}) & 2 \leq i \leq n \\ U_n(b_n, \vec{x_n}) & \to & r & \end{array}$$

where the U_i are fresh new symbols added to Σ^*, with the context-sensitive strategy $\mu^*(U_i) = \{1\}$. That is, only the first argument may be evaluated. The $\vec{x_i}$ are vectors of variables defined as $\vec{x_i} = \mathrm{Var}(l) \cup \mathrm{Var}(b_1) \cup \cdots \cup \mathrm{Var}(b_{i-1})$ for $1 \leq i \leq n$, which, by admissibility, ensures that in the above rules each of right-hand side variable occurs in the left-hand side; or, in a clever way so as to avoid keeping track of unused variables:

$$\begin{array}{rcl} \vec{x_i} & = & (\mathrm{Var}(l) \cup \mathrm{Var}(b_1) \cup \cdots \cup \mathrm{Var}(b_{i-1})) \\ & \cap & (\mathrm{Var}(b_i) \cup \mathrm{Var}(a_{i+1}) \cup \mathrm{Var}(b_{i+1}) \cup \cdots \\ & & \cup \quad \mathrm{Var}(a_n) \cup \mathrm{Var}(b_n) \cup \mathrm{Var}(r)) \end{array}$$

EXAMPLE 3. *For our running example (take the optimized version in Section 3.1.2), the corresponding unconditional translation includes (we only show the rules stemming from conditional rules):*

```
op uTake1 : S -> S [strat (1 0)] .
op uTake2 : S S S S -> S [strat (1 0)] .
op uLength : S S -> S [strat (1 0)] .
eq take(0,IL) = uTake1(isNatIList(IL)) .
eq uTake1(tt) = nil .
eq take(s(M),cons(N,IL)) =
   uTake2(and(isNat(M),and(isNat(N),isNatIList(IL))),
          M,N,IL) .
eq uTake2(tt,M,N,IL) = cons(N,take(M,IL)) .
eq length(cons(N,L)) =
   uLength(and(isNat(N),isNatList(L)),L) .
eq uLength(tt,L) = s(length(L)) .
```

LEMMA 3. *For any ground terms t and u: if $(T, \mu) \vdash t \to^* u$ then $(T^*, \mu^*) \vdash t \to^* u$. Moreover if $(T, \mu) \vdash t \to^+ u$ then $(T^*, \mu^*) \vdash t \to^+ u$.*

PROOF. By induction on the size of the proof tree of $(T, \mu) \vdash t \to^* u$ the last proof step was either

- By (Reflexivity) or (Transitivity): we conclude $(T^*, \mu^*) \vdash t \to^* u$ also by (Reflexivity) or (Transitivity).
- By (Congruence) that is $t = f(t_1, \ldots, t_k)$, $u = f(u_1, \ldots, u_k)$, $\mu(f) = \{i_1 \ldots i_k\}$ and for each j, $(T, \mu) \vdash t_{i_j} \to^* u_{i_j}$. By induction, $(T^*, \mu^*) \vdash t_{i_j} \to^* u_{i_j}$ and since in the replacement map for f is kept the same, we conclude again by (Congruence).
- By (Replacement): there exists a rule $l \to r$ if $a_1 \to b_1, \ldots, a_n \to b_n$ in T and a substitution σ such that $t = l\sigma$,

$u = r\sigma$, and $(T, \mu) \vdash a_i\sigma \to^* b_i\sigma$. By induction, $(T^*, \mu^*) \vdash a_i\sigma \to^* b_i\sigma$. In T^*, there are the necessary rules to perform the reduction

$$\begin{array}{rcll} t = l\sigma & \to & U_1(a_1\sigma, \vec{x_1}\sigma) & \text{(Replacement)} \\ & \to^* & U_1(b_1\sigma, \vec{x_1}\sigma) & \text{(Congruence)} \\ & \to & U_2(a_2\sigma, \vec{x_2}\sigma) & \text{(Replacement)} \\ & \vdots & & \\ & \to^* & U_n(b_n\sigma, \vec{x_n}\sigma) & \text{(Congruence)} \\ & \to & r\sigma & \text{(Replacement)} \end{array}$$

Notice that the (Congruence) steps above are allowed because $1 \in \mu^*(U_i)$.

If $(T, \mu) \vdash t \to^+ u$, then it is straightforward to see that $(T^*, \mu^*) \vdash t \to^+ u$ by examining each of the four cases above. □

THEOREM 3. *If the system (T^*, μ^*) resulting from transformation of (T, μ) is terminating, then (T, μ) is also terminating.*

PROOF. Any infinite reduction for (T, μ) could be lifted into an infinite reduction for (T^*, μ^*), using the above lemma. □

EXAMPLE 4. *According to Theorems 3 and 2, termination of program* `OvConsOS` *in Example 1 can be proved by proving the μ-termination of the following Term Rewriting System:*

```
and(tt,T) -> T
isNatIList(IL) -> isNatList(IL)
isNat(0) -> tt
isNat(s(N)) -> isNat(N)
isNat(length(L)) -> isNatList(L)
isNatIList(zeros) -> tt
isNatIList(cons(N,IL)) -> and(isNat(N),isNatIList(IL))
isNatList(nil) -> tt
isNatList(cons(N,L)) -> and(isNat(N),isNatList(L))
isNatList(take(N,IL)) -> and(isNat(N),isNatIList(IL))
zeros -> cons(0,zeros)
take(0,IL) -> uTake1(isNatIList(IL))
uTake1(tt) -> nil
take(s(M),cons(N,IL)) ->
   uTake2(and(isNat(M),and(isNat(N),isNatIList(IL))),
   M,N,IL)
uTake2(tt,M,N,IL) -> cons(N,take(M,IL))
length(cons(N,L)) ->
   uLength(and(isNat(N),isNatList(L)),L)
uLength(tt,L) -> s(length(L))
```

where $\mu(\texttt{isNat}) = \mu(\texttt{isNatList}) = \mu(\texttt{isNatIList}) = \emptyset$, $\mu(\texttt{cons}) = \mu(\texttt{uTake1}) = \mu(\texttt{uTake2}) = \mu(\texttt{uLength}) = \{1\}$ *and* $\mu(f) = \{1, \ldots, ar(f)\}$ *for all other symbols* f.

4. FROM THEORY TO PRACTICE

As remarked in the introduction, once we have obtained a CS-TRS (i.e., a TRS R together with a replacement map μ), we can just try a proof of μ-termination of R (i.e., termination of *CSR* for R and the replacement map μ). Fortunately, several methods have been developed for this purpose. There exist some *direct methods* for proving termination of *CSR*. These are orderings $>$ on terms which can be used to directly compare the left-hand sides and right-hand sides of the rules in order to conclude the μ-termination of the TRS [4, 22]. The MU-TERM tool is able to give a direct and automatic proof of termination of *CSR* by using the polynomial interpretations of [22].

EXAMPLE 5. *The TRS R in Figure 7 can be used to approximate the value of $\pi/2$ by means of the so-called* Wallis' product*: $\frac{\pi}{2} = lim_{n\to\infty} \frac{2}{1}\frac{2}{3}\frac{4}{3}\frac{4}{5}\cdots\frac{2n}{2n-1}\frac{2n}{2n+1}$. The expression*

```
take(s(N),cons(X,XS)) → cons(X,take(N,XS))
take(0,XS) → nil
incr(cons(X,XS)) → cons(s(X),incr(XS))
pairNs → cons(0,incr(oddNs))
oddNs → incr(pairNs)
zip(nil,XS) → nil
zip(X,nil) → nil
zip(cons(X,XS),cons(Y,YS)) → cons(pair(X,Y),zip(XS,YS))
tail(cons(X,XS)) → XS
repItems(cons(X,XS)) → cons(X,cons(X,repItems(XS)))
repItems(nil) → nil
```

Figure 7: Computing Wallis' approximation

```
zip(repItems(tail(pairNs)),tail(repItems(oddNs)))
```

produces the previous fractions and the function `take` *in Example 1 can be used to obtain appropriate approximations.*

Let $\mu(\texttt{cons}) = \{1\}$ and $\mu(f) = \{1, \ldots, ar(f)\}$ for all other symbols f. The μ-termination of R can also be proved by the following polynomial interpretation (computed by MU-TERM *with the help of* C*i*ME*):*

```
[nil] = 1                          [tail](X) = 4.X
[pairNs] = 3                       [repItems](X) = 2.X
[cons](X1,X2) = X1 + 1/4.X2 + 1    [0] = 0
[pair](X1,X2) = X1 + X2            [oddNs] = 5
[zip](X1,X2) = 2.X1 + 2.X2         [s](X) = X
[take](X1,X2) = X1 + X2 + 2        [incr](X) = X + 1
```

Unfortunately, regarding our running example, it is not difficult to see that no polynomial interpretation can be used to prove termination of the TRS R in Example 4. However, it is not difficult to see that a potentially non-terminating expression like `length(zeros)` will not cause an infinite rewriting sequence in R due to the (unsuccessful) sort check performed for the expression `zeros` by the function `isNatList`.

On the other hand, [10, 13, 17, 32] describe a number of transformations Θ from TRSs R and replacement maps μ that produce TRSs R^μ_Θ. Then, if we are able to prove *termination* of R^μ_Θ (using the standard methods), termination of *CSR* under μ (i.e., the *μ-termination* of R) is ensured (see Section 4.2 below for a concrete example). MU-TERM also implements all the aforementioned transformations and provides interfaces for the external use of existing tools for proving termination (of R^μ_Θ): AProVE [14], C*i*ME [9], TERMPTATION [5], and TTT [16].

4.1 A Prototype Implementation

Our current prototype tool takes MAUDE equational programs as inputs and implements the transformations A and B described in Section 3. The tool implementation clearly distinguishes two parts: (1) the theory transformations described in the paper, which are performed inside MAUDE, and (2) a Java application connecting MAUDE and MU-TERM and providing a GUI. The Java application is in charge of sending the MAUDE specification introduced by the user to MAUDE to perform transformations; it then receives the result of the transformations and sends it to MU-TERM.

MAUDE's design and implementation systematically exploit the reflective capabilities of rewriting logic [8] through its built-in `META-LEVEL` module. This allows *meta-representing theories as data*, so that theory transformations become *equationally defined functions* on such meta-representations built on top of

```
(fmod MYNAT is
 sort Nat .
 op 0 : -> Nat .
 op s : Nat -> Nat .
 endfm)

(fmod LazyLNat is
 pr NAT .
 sort LNat .
 sort PLNat .
 op nil : -> LNat .
 op cons : Nat LNat -> LNat [strat (1 0)] .
 op pair : LNat LNat -> PLNat .
 op natsFrom : Nat -> LNat .
 op fst : PLNat -> LNat .
 op snd : PLNat -> LNat .
 var N : Nat .
 vars X Y : LNat .
 eq natsFrom(N) =
       cons(N, natsFrom(s(N))) .
 eq fst(pair(X, Y)) = X .
 eq snd(pair(X, Y)) = Y .
 endfm)

(fmod SplitAt is
 pr LazyLNat .
 op splitAt : Nat LNat -> PLNat .
 vars N X : Nat .
 vars XS YS ZS : LNat .
 eq splitAt(0, XS) = pair(nil, XS) .
 ceq splitAt(s(N), cons(X, XS)) =
   pair(cons(X, YS), ZS)
   if pair(YS, ZS) := splitAt(N, XS) .
 endfm)

(fmod ListUtilities is
 pr SplitAt .
 op head : LNat -> Nat .
 op tail : LNat -> LNat .
 op sel  : Nat LNat -> Nat .
 op take : Nat LNat -> LNat .
 op afterNth : Nat LNat -> LNat .
 vars N : Nat .
 vars XS : LNat .
 eq head(cons(N, XS)) = N .
 eq tail(cons(N, XS)) = XS .
 eq sel(N, XS) = head(afterNth(N, XS)) .
 eq take(N, XS) = fst(splitAt(N, XS)) .
 eq afterNth(N, XS) = snd(splitAt(N, XS)) .
 endfm)
```

Figure 8: Lazy lists example

META-LEVEL. The very abstract level at which the transformations are thus specified allows us to keep their implementation very close to their mathematical definitions, affording a greater flexibility, maintainability, and extensibility. The transformations have been developed as an extension of FULL MAUDE which, in addition to providing some significant infrastructure that facilitates the development, make it possible to access the transformations from other applications while using the power of the FULL MAUDE specifications. FULL MAUDE is a language extension of MAUDE written in MAUDE, that endows MAUDE with a powerful and extensible module algebra in which MAUDE modules can be combined together to build more complex modules [8].

4.2 Another example

We illustrate our techniques on the MAUDE program of Figure 8, which provides support for the use of 'lazy' lists of natural numbers. The local strategy $(1\ 0)$ for the list constructor '`cons`' (in module `LazyLNat`) freezes the second argument, thus allowing dealing with infinite lists such as the infinite sequence of natural numbers $0, 1, 2, \ldots$ which is obtained by evaluating `natsFrom(0)`. In module `ListUtilities`, we use the function `splitAt` for defining several functions for list manipulation. The function is defined in module `SplitAt` by using a conditional equation (borrowing the current definition in the prelude of the lazy functional language Haskell).

In this case, we use the simpler variant of transformation A that forgets sort information (see Section 3.1.2) which turns out to be very useful. Then, after collapsing all sorts into a single one and applying transformation B, we obtain the following CS-TRS (which we call `SplitAtTR`):

```
natsFrom(N) → cons(N, natsFrom(s(N)))
splitAt(0, XS) → pair(nil, XS)
splitAt(s(N), cons(X, XS)) → u(splitAt(N, XS), X)
u(pair(YS, ZS), X) → pair(cons(X, YS), ZS)
sel(N, XS) → head(afterNth(N, XS))
take(N, XS) → fst(splitAt(N, XS))
afterNth(N, XS) → snd(splitAt(N, XS))
fst(pair(XS, YS)) → XS
snd(pair(XS, YS)) → YS
head(cons(N, XS)) → N
tail(cons(N, XS)) → XS
```

where $\mu(\texttt{cons}) = \mu(u) = \{1\}$ and $\mu(f) = \{1, \ldots, ar(f)\}$ for all other symbols f in the signature.

This CS-TRS is proved μ-terminating by using Zantema's transformation. Zantema's transformation *marks* the *non-replacing arguments* of function symbols (disregarding their positions within the term) [32]. New function symbols are used to block further reductions at this position. In addition, if a variable x occurs in a non-replacing position in the lhs l of a rewrite rule $l \to r$, then all occurrences of x in r are replaced by `activate(`x`)`. Here, `activate` is a new unary function symbol which is used to activate blocked function symbols again. New rewrite rules are added for blocking and unblocking function symbols. For instance, if we apply Zantema's transformation[1] to `SplitAtTR`, we obtain the following TRS `SplitAtTR_Z` (which has been obtained by using MU-TERM):

```
natsFrom(N) → cons(N, n__natsFrom(s(N)))
fst(pair(XS, YS)) → XS
snd(pair(XS, YS)) → YS
splitAt(0, XS) → pair(nil, XS)
splitAt(s(N), cons(X, XS)) →
                      u(splitAt(N, activate(XS)), X)
u(pair(YS, ZS), X) → pair(cons(activate(X), YS), ZS)
head(cons(N, XS)) → N
tail(cons(N, XS)) → activate(XS)
sel(N, XS) → head(afterNth(N, XS))
take(N, XS) → fst(splitAt(N, XS))
afterNth(N, XS) → snd(splitAt(N, XS))
natsFrom(X) → n__natsFrom(X)
activate(n__natsFrom(X)) → natsFrom(X)
activate(X) → X
```

This is a standard TRS which can be proved terminating by C*i*ME. MU-TERM sends the system to C*i*ME to obtain the proof (see Figure 9).

Hence, by Theorem 3, the original program consisting of modules `LazyLNat`, `SplitAt` and `ListUtilities` is also terminating.

[1] We use the simple description of Zantema's transformation as given in [13].

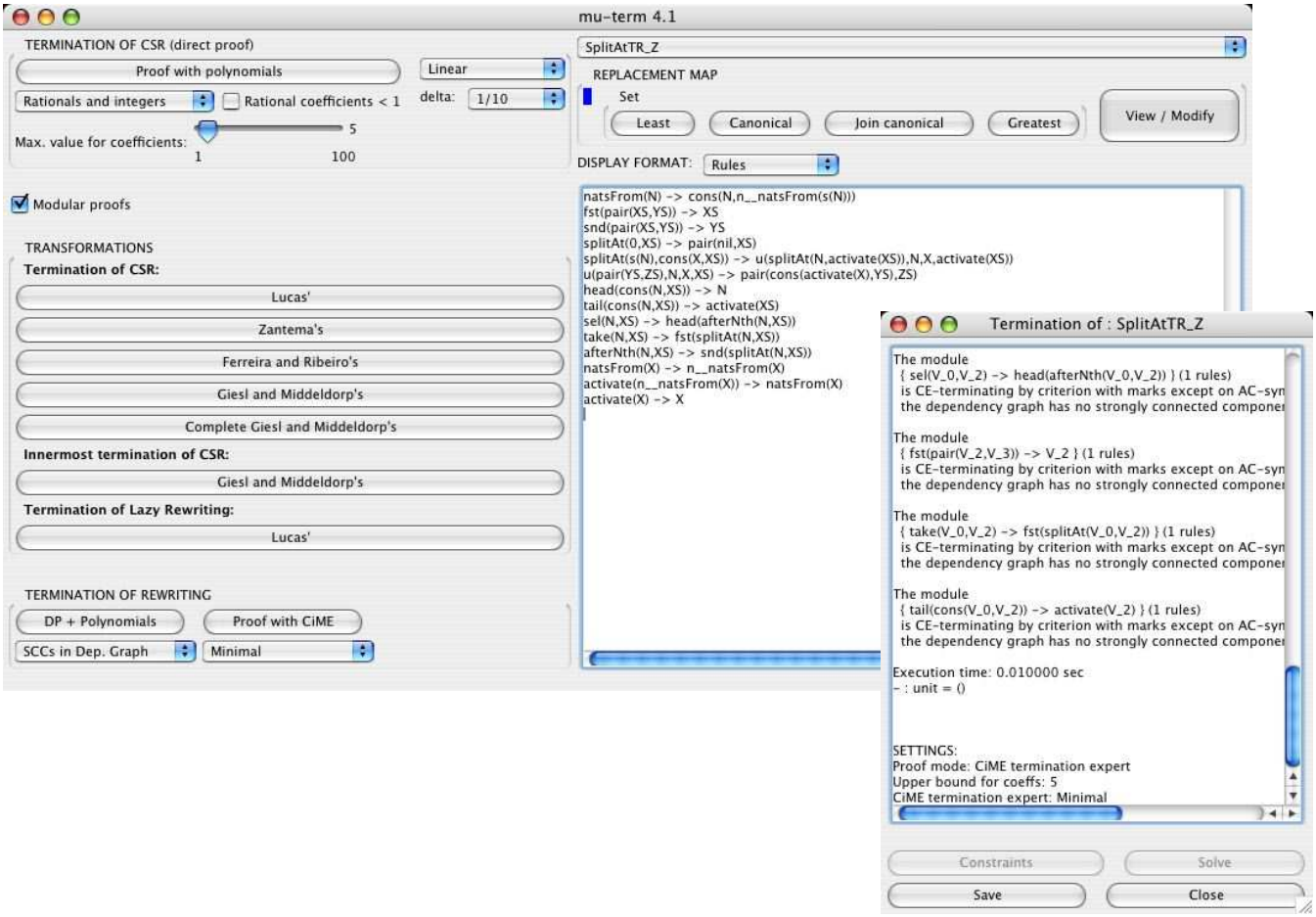

Figure 9: Termination of `SplitAtTR_Z` with MU-TERM and C*i*ME

5. CONCLUSIONS AND FURTHER WORK

Proving termination of equational programs having expressive features such as matching, typing, and evaluation strategies is important but nontrivial, because some of those features may not be supported by standard termination methods and tools. Yet, use of the features may be essential to ensure termination. We have presented a sequence of theory transformations that can be used to bridge the gap between equational programs and termination tools, have proved their correctness, and have discussed a prototype implementation in a tool taking MAUDE functional modules as inputs, performing the transformations, and mapping the resulting transformed theories to MU-TERM and from there to C*i*ME, and other termination tools. Much work remains ahead, both in theoretical aspects and in experimentation. Theoretical issues that need to be further investigated include the following.

First, extending our methods to prove termination of equational programs with *innermost* contextual rewriting in the case of *conditional rules*. For unconditional specifications, methods for such termination already exists and have been shown equivalent to proving termination of programs with elementary E-strategies in the OBJ sense [20]. There are also tools like APROVE or TERMPTATION which permit to prove termination of innermost rewriting; and there are also tools like CARIBOO [11] which are specialized in dealing with termination of rewriting under strategies (in particular, a class of innermost context-sensitive strategies). The main issue here is in fact how to define the reduction relation: for example, with the two rules $f(a) \to f(b)$ and $a \to b$ if $f(a) \to f(b)$, do we have $f(a) \to^* f(b)$ with innermost strategy? In an interpreter like MAUDE, asking normalization of a loops forever because it tries to apply the second rule, hence tries to reduce $f(a)$ innermost, hence tries to normalize a again. This issue is indeed a particular case of the more general issue of dealing with *effective termination*, that is termination of the computation on all queries given to an interpreter. For example, the one-rule CTRS $a \to b$ if $a \to c$ is terminating in the sense given in this paper, since the relation $\to^+$ is empty, but asking normalization of a would again loop forever.

Second, extending existing modular/incremental termination proof techniques [1, 23, 25, 26, 28, 29] to our setting. Since MAUDE programs are built by composition of modules, termination should be proven incrementally: each time a new module is added, a proof of termination should be obtained by using the knowledge of termination of previous ones. However, further investigation is required, since MAUDE module hierarchies do not necessarily respect the usual hierarchical property required for hierarchical TRSs, namely that for each rule added, the left-hand side's root symbol is a new symbol. Furthermore, even if this were to hold for some MAUDE programs, the transformations we have defined do not preserve that property, in particular because of sort elimination: if a new symbol f declares an old sort S as its codomain,

then a new rule $is_S(f(\ldots)) \to \ldots$ is added, whereas is_S is an old symbol.

Third, extending the methods of this paper from equational theories to theories in rewriting logic that allow frozen arguments when rewriting with rules [7]. This amounts to considering two levels of contextual rewriting: one for rewriting with equational rules according to some strategy μ, and another for rewriting with nonequational rules with another strategy μ'. In practice this will allow proving termination of concurrent systems specified as rewrite theories.

More experimentation is needed to further extend and refine our methods. The current prototype provides a first basis for such experimentation; it should be extended and improved in several directions, including, adding interfaces to other equational languages and termination tools, and adding support for the theoretical extensions mentioned above.

6. REFERENCES

[1] T. Arts, J. Giesl, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symb. Comp.*, 34(2):21–58, 2002.

[2] M. Bidoit and P. Mosses. *CASL User Manual*, volume 2900 of *LNCS*. Springer-Verlag, 2004.

[3] 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.

[4] C. Borralleras, S. Lucas, and A. Rubio. Recursive path orderings can be context-sensitive. In A. Voronkov, editor, *Proc. of 18th International Conference on Automated Deduction*, volume 2392 of *LNAI*, pages 314–331. Springer-Verlag, 2002.

[5] C. Borralleras and A. Rubio. Termptation. In Rubio [27], pages 61–63. Technical Report DSIC II/15/03, Universidad Politécnica de Valencia, Spain.

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

[7] R. Bruni and J. Meseguer. Generalized rewrite theories. In J. Baeten, J. Lenstra, J. Parrow, and G. Woeginger, editors, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 252–266, 2003.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Comput. Sci.*, 285(2):187–243, Aug. 2002.

[9] E. Contejean, C. Marché, B. Monate, and X. Urbain. Proving termination of rewriting with C*i*ME. In Rubio [27]. `http://cime.lri.fr`.

[10] M. Ferreira and A. Ribeiro. Context-sensitive ac-rewriting. In P. Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *LNCS*, pages 286–300, Trento, Italy, July 1999. Springer-Verlag.

[11] O. Fissore, I. Gnaedig, and H. Kirchner. Cariboo: An induction based proof tool for termination with strategies. In C. Kirchner, editor, *4th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, Pittsburgh, USA, Oct. 2002. ACM Press.

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

[13] J. Giesl and A. Middeldorp. Transformation techniques for context-sensitive rewrite systems. *Journal of Functional Programming*, 2004. To appear.

[14] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. AProVE: A system for proving termination. In Rubio [27]. `http://www-i2.informatik.rwth-aachen.de/AProVE`.

[15] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000.

[16] N. Hirokawa and A. Middeldorp. Tsukuba termination tool. In R. Nieuwenhuis, editor, *14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *LNCS*, pages 311–320, Valencia, Spain, June 2003. Springer-Verlag.

[17] S. Lucas. Termination of context-sensitive rewriting by rewriting. In F. M. auf der Heide and B. Monien, editors, *Proc. of ICALP'96*, volume 1099 of *LNCS*, pages 122–133. Springer-Verlag, 1996.

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

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

[20] S. Lucas. Termination of programs with strategy annotations. Technical Report DSIC-II/20/03, DSIC, Universidad Politécnica de Valencia, September 2003.

[21] S. Lucas. MU-TERM, a tool for proving termination of context-sensitive rewriting. In V. van Oostrom, editor, *15th International Conference on Rewriting Techniques and Applications*, LNCS, Aachen, Germany, June 2004. Springer-Verlag. `http://www.dsic.upv.es/~slucas/csr/termination/muterm/`.

[22] S. Lucas. Polynomials for proving termination of context-sensitive rewriting. In I. Walukiewicz, editor, *Proc. of 7th International Conference on Foundations of Software Science and Computation Structures, FOSSACS'04*, volume 2987 of *LNCS*, pages 318–332. Springer-Verlag, 2004.

[23] C. Marché and X. Urbain. Modular & incremental proofs of AC-termination. *Journal of Symb. Comp.*, 2004.

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

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

[26] E. Ohlebusch. Hierarchical termination revisited. *Inf. Process. Lett.*, 84(4):207–214, 2002.

[27] A. Rubio, editor. *6th International Workshop on Termination, WST'03*, June 2003. Technical Report DSIC II/15/03, Universidad Politécnica de Valencia, Spain.

[28] X. Urbain. Automated incremental termination proofs for hierarchically defined term rewriting systems. In R. Goré, A. Leitsch, and T. Nipkow, editors, *IJCAR'01*, volume 2083 of *LNAI*, pages 485–498, Siena, Italy, June 2001. Springer-Verlag.

[29] X. Urbain. Modular and incremental automated termination proofs. *Journal of Automated Reasoning*, 2004.

[30] A. van Deursen, J. Heering, and P. Klint. *Language Prototyping: An Algebraic Specification Approach*. World Scientific, 1996.

[31] H. Zantema. Termination of term rewriting: interpretation

and type elimination. *Journal of Symb. Comp.*, 17:23–50, 1994.

[32] H. Zantema. Termination of context-sensitive rewriting. In H. Comon, editor, *8th International Conference on Rewriting Techniques and Applications*, volume 1232 of *LNCS*, pages 172–186, Sitges, Spain, June 1997. Springer-Verlag.