# Computing More Specific Versions of Conditional Rewriting Systems *

Naoki Nishida[1] and Germán Vidal[2]

[1] Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan,
nishida@is.nagoya-u.ac.jp
[2] MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
gvidal@dsic.upv.es

**Abstract.** Rewrite systems obtained by some automated transformation often have a poor syntactic structure even if they have good properties from a semantic point of view. For instance, a rewrite system might have overlapping left-hand sides even if it can only produce at most one constructor normal form (i.e., value). In this paper, we propose a method for computing "more specific" versions of deterministic conditional rewrite systems (i.e., typical functional programs) by replacing a given rule (e.g., an overlapping rule) with a finite set of instances of this rule. In some cases, the technique is able to produce a non-overlapping system from an overlapping one. We have applied the transformation to improve the systems produced by a previous technique for function inversion with encouraging results (all the overlapping systems were successfully transformed to non-overlapping systems).

## 1 Introduction

Rewrite systems [4] form the basis of several rule-based programming languages. In this work, we focus on the so called *deterministic* conditional rewrite systems (DCTRSs), which are typical functional programs with local declarations [23]. When the rewrite systems are automatically generated (e.g., by program inversion [2, 14, 15, 17, 22, 27, 26, 28] or partial evaluation [1, 7, 8, 35]), they often have a poor syntactic structure that might hide some properties. For instance, the rewriting systems generated by program inversion sometimes have overlapping left-hand sides despite the fact that they actually have the *unique normal form* property w.r.t. constructor terms—i.e., they can only produce at most one constructor normal form for every expression—or are even confluent.

* This work has been partially supported by the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant PROMETEO/2011/052, and by *MEXT KAKENHI* #21700011.

Consider, e.g., the following TRS (left) from [28] (where we use $[\_|\_]$ and $\mathsf{nil}$ as list constructors) and its inversion (right):

$$\begin{array}{ll} \mathsf{inc}(\mathsf{nil}) \to [\mathsf{0}] & \mathsf{inc}^{-1}([\mathsf{0}]) \to \mathsf{nil} \\ \mathsf{inc}([\mathsf{0}|xs]) \to [\mathsf{1}|xs] & \mathsf{inc}^{-1}([\mathsf{1}|xs]) \to [\mathsf{0}|xs] \\ \mathsf{inc}([\mathsf{1}|xs]) \to [\mathsf{0}|\mathsf{inc}(xs)] & \mathsf{inc}^{-1}([\mathsf{0}|ys]) \to [\mathsf{1}|\mathsf{inc}^{-1}(ys)] \end{array}$$

Now, observe that every instance of $\mathsf{inc}^{-1}(x)$ using a constructor term has a unique constructor normal form. However, this system is not confluent—consider, e.g., the reductions starting from $\mathsf{inc}^{-1}([\mathsf{0}])$—and, moreover some of the left-hand sides overlap, thus preventing us from obtaining a typical (deterministic) functional program. In this case, one can observe that the recursive call to $\mathsf{inc}^{-1}$ in the third rule can only bind variable $ys$ to a non-empty list, say $[z|zs]$. Therefore, the system above could be transformed as follows:

$$\begin{array}{l} \mathsf{inc}^{-1}([\mathsf{0}]) \to \mathsf{nil} \\ \mathsf{inc}^{-1}([\mathsf{1}|xs]) \to [\mathsf{0}|xs] \\ \mathsf{inc}^{-1}([\mathsf{0}, z|zs]) \to [\mathsf{1}|\mathsf{inc}^{-1}([z|zs])] \end{array}$$

Now, the transformed system is non-overlapping (and confluent) and, under some conditions, it is equivalent to the original system (roughly speaking, the derivations to constructor terms are the same).

A "more specific" transformation was originally introduced by Marriott et al. [24] in the context of logic programming. In this context, a *more specific version* of a logic program is a version of this program where each clause is further instantiated or removed while preserving the successful derivations of the original program. According to [24], the transformation increases the number of finitely failed goals (i.e., some infinite derivations are transformed into finitely failed ones), detects failure more quickly, etc. In general, the information about the allowed variable bindings which is hidden in the original program may be made explicit in a more specific version, thus improving the static analysis of the program's properties.

In this paper, we adapt the notion of a more specific program to the context of deterministic conditional term rewriting systems. In principle, the transformation may achieve similar benefits as in logic programming by making some variable bindings explicit (as illustrated in the transformation of function $\mathsf{inc}^{-1}$ above). Adapting this notion to rewriting systems, however, is far from trivial:

- First, there is no clear notion of *successful* reduction, and aiming at preserving all possible reductions would give rise to a meaningless notion. Consider, e.g., the rewrite systems $\mathcal{R} = \{\mathsf{f}(x) \to \mathsf{g}(x),\ \mathsf{g}(\mathsf{a}) \to \mathsf{b}\}$ and $\mathcal{R}' = \{\mathsf{f}(\mathsf{a}) \to \mathsf{g}(\mathsf{a}),\ \mathsf{g}(\mathsf{a}) \to \mathsf{b}\}$. Here, one would expect $\mathcal{R}'$ to be a correct more specific version of $\mathcal{R}$ (in the sense of $\to_{\mathcal{R}} = \to_{\mathcal{R}'}$). However, the reduction $\mathsf{f}(\mathsf{b}) \to_{\mathcal{R}} \mathsf{g}(\mathsf{b})$ is not possible in $\mathcal{R}'$.
- Second, different reduction *strategies* require different conditions in order to guarantee the correctness of the transformation.
- Finally, in contrast to [24], we often require computing a set of instances of a rewrite rule (rather than a single, more general instance) in order to produce

non-overlapping left-hand sides. Consider the rewrite system $\mathcal{R} = \{\mathsf{f}(\mathsf{a}) \to \mathsf{a},\ \mathsf{f}(x) \to \mathsf{g}(x),\ \mathsf{g}(\mathsf{b}) \to \mathsf{b},\ \mathsf{g}(\mathsf{c}) \to \mathsf{c}\}$. If we aim at computing a single more specific version for the second rule, only the same rule is obtained. However, if a set of instances is allowed rather a single common instance, we can obtain the system $\mathcal{R}' = \{\mathsf{f}(\mathsf{a}) \to \mathsf{a},\ \mathsf{f}(\mathsf{b}) \to \mathsf{g}(\mathsf{b}),\ \mathsf{f}(\mathsf{c}) \to \mathsf{g}(\mathsf{c}),\ \mathsf{g}(\mathsf{b}) \to \mathsf{b},\ \mathsf{g}(\mathsf{c}) \to \mathsf{c}\}$, which is non-overlapping.

Apart from introducing the notions of *successful* reduction and *more specific* version (MSV), we provide a constructive algorithm for computing more specific versions, which is based on constructing finite (possibly incomplete) *narrowing* trees for the right-hand sides of the original rewrite rules. Here, narrowing [36, 19], an extension of term rewriting by replacing pattern matching with unification, is used to *guess* the allowed variable bindings for a given rewrite rule. We prove the correctness of the algorithm (i.e., that it actually outputs a more specific version of the input system). We have tested the usefulness of the MSV transformation to improve the inverse systems obtained by the program inversion method of [27, 26, 28], and our preliminary results are very encouraging.

This paper is organized as follows. In Section 2, we briefly review some notions and notations of term rewriting and narrowing. Section 3 introduces the notions of more specific version of a rewriting system. Then, we introduce an algorithm for computing more specific versions in Section 4 and prove its correctness. Finally, Section 5 concludes and points out some directions for future research. A preliminary version of this paper—with a more restricted notion of more specific version and where only unconditional rules are considered—appeared in [30].

## 2 Preliminaries

We assume familiarity with basic concepts of term rewriting and narrowing. We refer the reader to, e.g., [4], [32], and [16] for further details.

*Terms and Substitutions.* A *signature* $\mathcal{F}$ is a set of function symbols. Given a set of variables $\mathcal{V}$ with $\mathcal{F} \cap \mathcal{V} = \varnothing$, we denote the domain of *terms* by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We assume that $\mathcal{F}$ always contains at least one constant $\mathsf{f}/0$. We use $\mathsf{f}, \mathsf{g}, \ldots$ to denote functions and $x, y, \ldots$ to denote variables. Positions are used to address the nodes of a term viewed as a tree. A *position* $p$ in a term $t$ is represented by a finite sequence of natural numbers, where $\epsilon$ denotes the root position. We let $t|_p$ denote the *subterm* of $t$ at position $p$ and $t[s]_p$ the result of *replacing the subterm* $t|_p$ by the term $s$. $\mathcal{V}ar(t)$ denotes the set of variables appearing in $t$. A term $t$ is *ground* if $\mathcal{V}ar(t) = \varnothing$.

A *substitution* $\sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a mapping from variables to terms such that $\mathcal{D}\mathsf{om}(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is its domain. Substitutions are extended to morphisms from $\mathcal{T}(\mathcal{F}, \mathcal{V})$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ in the natural way. We denote the application of a substitution $\sigma$ to a term $t$ by $t\sigma$ rather than $\sigma(t)$. The identity substitution is denoted by *id*. A *variable renaming* is a substitution that is a bijection on $\mathcal{V}$. A substitution $\sigma$ is *more general* than a substitution $\theta$, denoted by $\sigma \leqslant \theta$, if there is a substitution $\delta$ such that $\delta \circ \sigma = \theta$, where "$\circ$" denotes the composition of substitutions (i.e., $\sigma \circ \theta(x) = (x\theta)\sigma = x\theta\sigma$). The *restriction*

$\theta\restriction_V$ of a substitution $\theta$ to a set of variables $V$ is defined as follows: $x\theta\restriction_V = x\theta$ if $x \in V$ and $x\theta\restriction_V = x$ otherwise. We say that $\theta = \sigma\ [V]$ if $\theta\restriction_V = \sigma\restriction_V$.

A term $t_2$ is an *instance* of a term $t_1$ (or, equivalently, $t_1$ is *more general* than $t_2$), in symbols $t_1 \leqslant t_2$, if there is a substitution $\sigma$ with $t_2 = t_1\sigma$. Two terms $t_1$ and $t_2$ are *variants* (or equal up to variable renaming) if $t_1 = t_2\rho$ for some variable renaming $\rho$. A *unifier* of two terms $t_1$ and $t_2$ is a substitution $\sigma$ with $t_1\sigma = t_2\sigma$; furthermore, $\sigma$ is the *most general unifier* of $t_1$ and $t_2$, denoted by $\mathsf{mgu}(t_1, t_2)$ if, for every other unifier $\theta$ of $t_1$ and $t_2$, we have that $\sigma \leqslant \theta$.

*TRSs and Rewriting.* A set of rewrite rules $l \to r$ such that $l$ is a nonvariable term and $r$ is a term whose variables appear in $l$ is called a *term rewriting system* (TRS for short); terms $l$ and $r$ are called the left-hand side and the right-hand side of the rule, respectively. We restrict ourselves to finite signatures and TRSs. Given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the *defined* symbols $\mathcal{D}_\mathcal{R}$ are the root symbols of the left-hand sides of the rules and the *constructors* are $\mathcal{C}_\mathcal{R} = \mathcal{F} \setminus \mathcal{D}_\mathcal{R}$. *Constructor terms* of $\mathcal{R}$ are terms over $\mathcal{C}_\mathcal{R}$ and $\mathcal{V}$. We sometimes omit $\mathcal{R}$ from $\mathcal{D}_\mathcal{R}$ and $\mathcal{C}_\mathcal{R}$ if it is clear from the context. A substitution $\sigma$ is a *constructor substitution* (of $\mathcal{R}$) if $x\sigma \in \mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$ for all variables $x$.

For a TRS $\mathcal{R}$, we define the associated rewrite relation $\to_\mathcal{R}$ as the smallest binary relation satisfying the following: given terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have $s \to_\mathcal{R} t$ iff there exist a position $p$ in $s$, a rewrite rule $l \to r \in \mathcal{R}$ and a substitution $\sigma$ with $s|_p = l\sigma$ and $t = s[r\sigma]_p$; the rewrite step is often denoted by $s \to_{p,l\to r} t$ to make explicit the position and rule used in this step. The instantiated left-hand side $l\sigma$ is called a *redex*.

A term $t$ is called *irreducible* or in *normal form* w.r.t. a TRS $\mathcal{R}$ if there is no term $s$ with $t \to_\mathcal{R} s$. A substitution is called *normalized* w.r.t. $\mathcal{R}$ if every variable in the domain is replaced by a normal form w.r.t. $\mathcal{R}$. We sometimes omit "w.r.t. $\mathcal{R}$" if it is clear from the context. We denote the set of normal forms by $NF_\mathcal{R}$. A *derivation* is a (possibly empty) sequence of rewrite steps. Given a binary relation $\to$, we denote by $\to^*$ its reflexive and transitive closure. Thus $t \to^*_\mathcal{R} s$ means that $t$ can be reduced to $s$ in $\mathcal{R}$ in zero or more steps; we also use $t \to^n_\mathcal{R} s$ to denote that $t$ can be reduced to $s$ in exactly $n$ rewrite steps.

A conditional TRS (CTRS) is a set of rewrite rules $l \to r \Leftarrow C$, where $C$ is a set of equations. In particular, we consider only *oriented* equations of the form $s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$. For a CTRS $\mathcal{R}$, we define the associated rewrite relation $\to_\mathcal{R}$ as follows: given terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have $s \to_\mathcal{R} t$ iff there exist a position $p$ in $s$, a rewrite rule $l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n \in \mathcal{R}$ and a substitution $\sigma$ such that $s|_p = l\sigma$, $s_i\sigma \to^*_\mathcal{R} t_i\sigma$ for all $i = 1, \ldots, n$, and $t = s[r\sigma]_p$.

*Narrowing.* The *narrowing* principle [36] mainly extends term rewriting by replacing pattern matching with unification, so that terms containing logic (i.e., *free*) variables can also be reduced by non-deterministically instantiating these variables. Conceptually, this is not significantly different from ordinary rewriting when TRSs contain *extra variables* (i.e., variables that appear in the right-hand side of a rule but not in its left-hand side), as noted in [3]. Formally, given a

TRS $\mathcal{R}$ and two terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have that $s \leadsto_{\mathcal{R}} t$ is a *narrowing step* iff there exist[3]

- a nonvariable position $p$ of $s$,
- a variant $l \to r$ of a rule in $\mathcal{R}$,
- a substitution $\sigma = \mathsf{mgu}(s|_p, l)$ which is a most general unifier of $s|_p$ and $l$,

and $t = (s[r]_p)\sigma$. We often write $s \leadsto_{p,l\to r,\theta} t$ (or simply $s \leadsto_\theta t$) to make explicit the position, rule, and substitution of the narrowing step, where $\theta = \sigma\restriction_{\mathcal{V}\mathsf{ar}(s)}$ (i.e., we label the narrowing step only with the bindings for the narrowed term). A *narrowing derivation* $t_0 \leadsto^*_\sigma t_n$ denotes a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$). Given a narrowing derivation $s \leadsto^*_\sigma t$ with $t$ a constructor term, we say that $\sigma$ is a *computed answer* for $s$.

*Example 1.* Consider the TRS

$$\mathcal{R} = \left\{ \begin{array}{ll} \mathsf{add}(\mathsf{0}, y) \to y & (R_1) \\ \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) & (R_2) \end{array} \right\}$$

defining the addition $\mathsf{add}/2$ on natural numbers built from $\mathsf{0}/0$ and $\mathsf{s}/1$. Given the term $\mathsf{add}(x, \mathsf{s}(\mathsf{0}))$, we have infinitely many narrowing derivations starting from $\mathsf{add}(x, \mathsf{s}(\mathsf{0}))$, e.g.,

$$\begin{array}{ll} \mathsf{add}(x, \mathsf{s}(\mathsf{0})) \leadsto_{\epsilon, R_1, \{x \mapsto \mathsf{0}\}} & \mathsf{s}(\mathsf{0}) \\ \mathsf{add}(x, \mathsf{s}(\mathsf{0})) \leadsto_{\epsilon, R_2, \{x \mapsto \mathsf{s}(y_1)\}} & \mathsf{s}(\mathsf{add}(y_1, \mathsf{s}(\mathsf{0}))) \leadsto_{1, R_1, \{y_1 \mapsto \mathsf{0}\}} \mathsf{s}(\mathsf{s}(\mathsf{0})) \\ \ldots & \end{array}$$

with computed answers $\{x \mapsto \mathsf{0}\}$, $\{x \mapsto \mathsf{s}(\mathsf{0})\}$, etc.

Narrowing is naturally extended to deal with equations and CTRSs (see Section 4).

## 3 More Specific Conditional Rewrite Systems

In this section, we introduce the notion of a *more specific* conditional rewrite system. Intuitively speaking, we produce a more specific CTRS $\mathcal{R}'$ from a CTRS $\mathcal{R}$ by replacing a conditional rewrite rule $(l \to r \Leftarrow C) \in \mathcal{R}$ with a *finite* number of *instances* of this rule, i.e., $\mathcal{R}' = (\mathcal{R} \setminus \{l \to r \Leftarrow C\}) \cup \{(l \to r \Leftarrow C)\sigma_1, \ldots, (l \to r \Leftarrow C)\sigma_n\}$, such that $\mathcal{R}'$ is *semantically* equivalent to $\mathcal{R}$ under some conditions.

The key idea is that more specific versions should still allow the same reductions of the original system. However, as mentioned in Section 1, if we aimed at preserving *all* possible rewrite reductions, the resulting notion would be useless since it would never happen in practice. Therefore, to have a practically applicable technique, we only aim at preserving what we call *successful* reductions. In the following, given a CTRS $\mathcal{R}$, we denote by $\xrightarrow{s}_{\mathcal{R}}$ a generic conditional rewrite relation based on some strategy $s$ (e.g., innermost conditional reduction $\xrightarrow{i}_{\mathcal{R}}$).

[3] We consider the so called *most general* narrowing, i.e., the $\mathsf{mgu}$ of the selected subterm and the left-hand side of a rule—rather than an arbitrary unifier—is computed at each narrowing step.

**Definition 1 (successful reduction w.r.t. $\xrightarrow{s}$).** *Let $\mathcal{R}$ be a CTRS and let $\xrightarrow{s}_{\mathcal{R}}$ be a conditional rewrite relation. A rewrite reduction $t \xrightarrow{s}{}^*_{\mathcal{R}} u$ where $t$ is a term and $u \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ is a constructor term, is called a* successful *reduction w.r.t. $\xrightarrow{s}_{\mathcal{R}}$.*

Let us now introduce our notion of a *more specific version* of a rewrite rule:

**Definition 2 (more specific version of a rule).** *Let $\mathcal{R}$ be a CTRS and $\xrightarrow{s}_{\mathcal{R}}$ be a conditional rewrite relation. Let $l \to r \Leftarrow C \in \mathcal{R}$ be a rewrite rule. We say that the finite set of rewrite rules $\mathcal{R}_{msv} = \{l_1 \to r_1 \Leftarrow C_1, \ldots, l_n \to r_n \Leftarrow C_n\}$ is a* more specific version of $l \to r \Leftarrow C$ in $\mathcal{R}$ w.r.t. $\xrightarrow{s}_{\mathcal{R}}$ *if*

- *there are substitutions $\sigma_1, \ldots, \sigma_n$ such that $(l \to r \Leftarrow C)\sigma_i = l_i \to r_i \Leftarrow C_i$ for $i = 1, \ldots, n$, and*
- *for all terms $t, u$, we have that $t \xrightarrow{s}{}^*_{\mathcal{R}} u$ is successful in $\mathcal{R}$ iff $t \xrightarrow{s}{}^*_{\mathcal{R}'} u$ is successful in $\mathcal{R}'$, with $\mathcal{R}' = (\mathcal{R} \setminus \{l \to r \Leftarrow C\}) \cup \mathcal{R}_{msv}$; moreover, we require $t \xrightarrow{s}{}^*_{\mathcal{R}} u$ and $t \xrightarrow{s}{}^*_{\mathcal{R}'} u$ to apply the same rules to the same positions and in the same order, except for the rule $l \to r \Leftarrow C$ in $\mathcal{R}$ that is replaced with some rule $l_i \to r_i \Leftarrow C_i$, $i \in \{1, \ldots, n\}$, in $\mathcal{R}'$.*[4]

Note that a rewrite rule is always a more specific version of itself, therefore the existence of a more specific version of a given rule is always ensured. In general, however, the more specific version is not unique, and thus, there can be several strictly more specific versions of a rewrite rule.

The notion of a more specific version of a rule is extended to CTRSs in a stepwise manner: given a CTRS $\mathcal{R}$, we first replace a rule of $\mathcal{R}$ by its more specific version thus producing $\mathcal{R}'$, then we replace another rule of $\mathcal{R}'$ by its more specific version, and so forth. We denote each of these steps by $\mathcal{R} \mapsto_{\mathsf{more}} \mathcal{R}'$. We say that $\mathcal{R}'$ is a *more specific version* of $\mathcal{R}$ if there is a sequence of (zero or more) $\mapsto_{\mathsf{more}}$ steps leading from $\mathcal{R}$ to $\mathcal{R}'$. Note that, given a CTRS $\mathcal{R}$ and one of its more specific versions $\mathcal{R}'$, we have that $\to_{\mathcal{R}'} \subseteq \to_{\mathcal{R}}$ (i.e., $NF_{\mathcal{R}} \subseteq NF_{\mathcal{R}'}$) and $\mathcal{D}_{\mathcal{R}} = \mathcal{D}_{\mathcal{R}'}$ (i.e., $\mathcal{C}_{\mathcal{R}} = \mathcal{C}_{\mathcal{R}'}$).

*Example 2.* Consider the following CTRS $\mathcal{R}$ (a fragment of a system obtained automatically by the function inversion technique of [28]):

$$\begin{aligned}
\mathsf{inv}([\mathsf{left}|x_2]) &\to (t, \mathsf{n}(x, y)) \Leftarrow \mathsf{inv}(x_2) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y) && (R_1)\\
\mathsf{inv}([\mathsf{str}(x)|t]) &\to (t, \mathsf{sym}(x)) && (R_2)\\
\mathsf{inv}([\mathsf{left}, \mathsf{right}|t]) &\to (t, \mathsf{bottom}) && (R_3)
\end{aligned}$$

where the definition of function $\mathsf{inv}'$ is not relevant for this example and we omit the tuple symbol (e.g., $\mathsf{tp}_2$) from the tuple of two terms—we write $(t_1, t_2)$ instead of $\mathsf{tp}_2(t_1, t_2)$. Here, we replace the first rule of $\mathcal{R}$ by the following two instances $\mathcal{R}_{msv}$:

$$\begin{aligned}
\mathsf{inv}([\mathsf{left}, \mathsf{left}|x_3]) &\to (t, \mathsf{n}(x, y)) \Leftarrow \mathsf{inv}([\mathsf{left}|x_2]) \twoheadrightarrow (x_1, x), \mathsf{inv}'(x_1) \twoheadrightarrow (t, y)\\
\mathsf{inv}([\mathsf{left}, \mathsf{str}(x_3)|x_4]) &\to (t, \mathsf{n}(x, y)) \Leftarrow \mathsf{inv}([\mathsf{str}(x_3)|x_4]) \twoheadrightarrow (x_1, x), \mathsf{inv}'(x_1) \twoheadrightarrow (t, y)
\end{aligned}$$

[4] This is required to prevent situations like the following one. Consider $\mathcal{R} = \{\mathsf{f}(\mathsf{0}) \to \mathsf{g}(\mathsf{0}),\ \mathsf{f}(x) \to \mathsf{g}(x),\ \mathsf{g}(\mathsf{0}) \to \mathsf{0}\}$. Here, any instance of rule $\mathsf{f}(x) \to \mathsf{g}(x)$ would be a more specific version if the last condition were not required (since the rule $\mathsf{f}(\mathsf{0}) \to \mathsf{g}(\mathsf{0})$ already belongs to $\mathcal{R}$).

using the substitutions $\sigma_1 = \{x_2 \mapsto [\mathsf{left}|x_3]\}$ and $\sigma_2 = \{x_2 \mapsto [\mathsf{str}(x_3)|x_4]\}$.

Observe that function $\mathsf{inv}$ in $(\mathcal{R}\backslash\{R_1\}) \cup \mathcal{R}_{msv}$ is now non-overlapping. Note that producing only a single instance would return an overlapping rule again since the only common generalization of $\sigma_1$ and $\sigma_2$ is $\{x_2 \mapsto [x_3|x_4]\}$ so that the more specific version would be

$$\mathsf{inv}([\mathsf{left}, x_3|x_4]) \to (t, \mathsf{n}(x, y)) \Leftarrow \mathsf{inv}([x_3|x_4]) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y)$$

and function $\mathsf{inv}$ would still be overlapping.

Now, we show a basic property of more specific versions of a CTRS. In the following, we say that a CTRS $\mathcal{R}$ *has the unique constructor normal form property w.r.t.* $\xrightarrow{s}_{\mathcal{R}}$ if, for all successful reductions $C, t \twoheadrightarrow x \xrightarrow{s}{}^* u \twoheadrightarrow x$ and $(C, t \twoheadrightarrow x) \xrightarrow{s}{}^* (u' \twoheadrightarrow x)$, with $u, u' \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, we have $u = u'$.

**Theorem 1.** *Let $\mathcal{R}$ be a CTRS and let $\mathcal{R}'$ be a more specific version of $\mathcal{R}$ w.r.t. $\xrightarrow{s}_{\mathcal{R}}$. Then, $\mathcal{R}$ has the unique constructor normal form property w.r.t. $\xrightarrow{s}_{\mathcal{R}}$ iff so does $\mathcal{R}'$.*

*Proof.* The claim follows straightforwardly since derivations producing a constructor normal form are successful derivations, and they are preserved by the MSV transformation by definition. □

## 4 Computing More Specific Versions

In this section, we tackle the definition of a constructive method for computing more specific versions of a CTRS. For this purpose, we consider the following assumptions:

- We restrict the class of CTRSs to *deterministic* CTRSs (DCTRSs [5, 13, 23]). Furthermore, we require them to be constructor systems (see below).
- We only consider constructor-based reductions (a particular case of innermost conditional reduction), i.e., reductions where the computed matching substitutions are constructor.
- We use a form of innermost conditional narrowing to approximate the potential successful reductions and, thus, compute its more specific version.

DCTRSs are *3-CTRSs* [5, 13, 23] (see also [32]) (i.e., CTRSs where extra variables are allowed as long as $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(C)$ for all rules $l \to r \Leftarrow C$), where the conditional rules have the form

$$l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$$

such that

- $\mathcal{V}\mathsf{ar}(s_i) \subseteq \mathcal{V}\mathsf{ar}(l) \cup \mathcal{V}\mathsf{ar}(t_1) \cup \cdots \cup \mathcal{V}\mathsf{ar}(t_{i-1})$, for all $i = 1, \ldots, n$;
- $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l) \cup \mathcal{V}\mathsf{ar}(t_1) \cup \ldots \cup \mathcal{V}\mathsf{ar}(t_n)$.

Moreover, we assume that the DCTRSs are constructor systems with $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$.

In DCTRSs, extra variables in the conditions are not problematic since no redex contains extra variables when it is selected. Actually, as noted by [23], these systems are basically equivalent to functional programs since every rule

$$l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$$

can be seen in a functional language as

$$\begin{array}{l} l \;=\; \textbf{let } t_1 = s_1 \textbf{ in} \\ \qquad\quad \textbf{let } t_2 = s_2 \textbf{ in} \\ \qquad\qquad \ldots \\ \qquad\qquad\quad \textbf{let } t_n = s_n \textbf{ in } r \end{array}$$

Here, DCTRSs allow us to represent functional *local definitions* using oriented conditions.

Under these conditions, innermost reduction extends quite naturally to the conditional case. In particular, we follow Bockmayr and Werner's *conditional rewriting without evaluation of the premise* [6], adapted to our setting as follows:

**Definition 3 (constructor reduction).** *Let $\mathcal{R}$ be a DCTRS. Constructor-based conditional reduction is defined as the smallest relation satisfying the following transition rules:*

$$\textit{(reduction)} \quad \frac{p = inn(s_1) \wedge l \to r \Leftarrow C \in \mathcal{R} \wedge s_1|_p = l\sigma \wedge \sigma \text{ is constructor}}{(s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n) \;\overset{c}{\to}\; (C\sigma, s_1[r\sigma]_p \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n)}$$

$$\textit{(matching)} \quad \frac{n > 1 \wedge s_1 \in \mathcal{T}(\mathcal{C}, \mathcal{V}) \wedge s_1 = t_1\sigma}{(s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n) \;\overset{c}{\to}\; (s_2 \twoheadrightarrow t_2, \ldots, s_n \twoheadrightarrow t_n)\sigma}$$

*where $inn(s)$ selects the position of an innermost subterm matchable with the left-hand side of a rule (i.e., a term $l'\sigma'$ of the form $f(c_1, \ldots, c_n)$ with $f \in \mathcal{D}$ and $c_1, \ldots, c_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ for some $l' \to r' \Leftarrow C' \in \mathcal{R}$ and some $\sigma'$), e.g., the leftmost one.*

Intuitively speaking, in order to reduce a sequence of equations $s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$, we consider two possibilities:

- If the first oriented equation has some innermost subterm that matches the left-hand side of a rewrite rule, we perform a reduction step. Note that, in contrast to ordinary conditional rewriting, the conditions are not verified but just added to the set of equations (as in Bockmayr and Werner's reduction without evaluation of the premise).
- If the left-hand side of the first oriented equation is a constructor term, then we match both sides (note that the right-hand side is a constructor term by definition) and remove this equation from the sequence. In the original definition of [6], this matching substitution is computed when applying a given rule in order to verify the conditions. Our definition simply makes computing the substitution more operational by postponing it to the point when its computation is required (and, thus, it is straightforward to compute).

Note that this rule requires having more than one equation, since the initial equation should not be removed.

In order to reduce a ground term $s$, we consider an initial oriented equation $s \twoheadrightarrow x$, where $x$ is a fresh variable not occurring in $s$, and reduce it as much as possible using the reduction and matching rules. If we reach an equation of the form $t \twoheadrightarrow x$, where $t$ is a constructor term, we say that $s$ reduces to $t$; actually, [6, Theorem 2] proves the equivalence between ordinary conditional rewriting and conditional rewriting without evaluation of the premise.

*Example 3.* Consider again the system $\mathcal{R}$ from Example 2 and the initial term $\mathsf{inv}([\mathsf{left}, \mathsf{str}(\mathsf{a})])$. We have, for instance, the following (incomplete) constructor-based reduction:

$$\begin{aligned}(\mathsf{inv}([\mathsf{left}, \mathsf{str}(\mathsf{a})]) &\twoheadrightarrow w)\\ &\xrightarrow{c} (\mathsf{inv}([\mathsf{str}(\mathsf{a})]) \twoheadrightarrow (x_1, x), \mathsf{inv}'(x_1) \twoheadrightarrow (t, y), (t, \mathsf{n}(x, y)) \twoheadrightarrow w)\\ &\xrightarrow{c} ((\mathsf{nil}, \mathsf{sym}(\mathsf{a})) \twoheadrightarrow (x_1, x), \mathsf{inv}'(x_1) \twoheadrightarrow (t, y), (t, \mathsf{n}(x, y)) \twoheadrightarrow w)\\ &\xrightarrow{c} (\mathsf{inv}'(\mathsf{nil}) \twoheadrightarrow (t, y), (t, \mathsf{n}(\mathsf{sym}(\mathsf{a}), y)) \twoheadrightarrow w)\end{aligned}$$

In the following, given a rule $l \to r \Leftarrow C$, we introduce the use of conditional narrowing to automatically compute an approximation of the successful constructor-based reductions.

Our definition of constructor-based conditional narrowing (a special case of innermost conditional narrowing [9, 12, 18]), denoted by $\overset{c}{\rightsquigarrow}$, is defined as follows:

**Definition 4 (constructor-based conditional narrowing).** *Let $\mathcal{R}$ be a DC-TRS. Constructor-based conditional narrowing is defined as the smallest relation satisfying the following transition rules:*

$$\textit{(narrowing)} \quad \frac{p = inn(s_1) \ \wedge \ l \to r \Leftarrow C \in \mathcal{R} \ \wedge \ \sigma = \mathsf{mgu}(s_1|_p, l)}{(s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n) \ \overset{c}{\rightsquigarrow}_\sigma \ (C, s_1[r]_p \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n)\sigma}$$

$$\textit{(unification)} \quad \frac{n > 1 \ \wedge \ s_1 \in \mathcal{T}(\mathcal{C}, \mathcal{V}) \ \wedge \ \sigma = \mathsf{mgu}(s_1, t_1)}{(s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n) \ \overset{c}{\rightsquigarrow}_\sigma \ (s_2 \twoheadrightarrow t_2, \ldots, s_n \twoheadrightarrow t_n)\sigma}$$

*where $inn(s)$ selects the position of an innermost subterm whose proper subterms are constructor terms, and which is unifiable with the left-hand side of a rule, e.g., the leftmost one.*

As can be seen, our definition of constructor-based conditional narrowing for DC-TRSs mimics the definition of constructor-based reduction but replaces matching with unification in both transition rules.

Note, however, that the first rule is often non-deterministic since a given innermost subterm can unify with the left-hand sides of several rewrite rules.

We adapt the notion of successful derivations to rewriting and narrowing derivations of equations.

**Definition 5.** *Let $\mathcal{R}$ be a CTRS and let $\overset{s}{\to}_\mathcal{R}$ be a conditional rewrite relation. A rewrite reduction $(C, t \twoheadrightarrow x) \overset{s}{\to}^*_\mathcal{R} (u \twoheadrightarrow x)$ where $t$ is a term, $x$ is a fresh variable, $C$ is a (possibly empty) sequence of (oriented) equations, and $u \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ is a constructor term, is called a* successful *reduction w.r.t. $\overset{s}{\to}_\mathcal{R}$.*

Note that $t \xrightarrow{s}{}^*_{\mathcal{R}} u$ is successful w.r.t. $\xrightarrow{s}_{\mathcal{R}}$ iff so is $(t \twoheadrightarrow x) \xrightarrow{s}{}^*_{\mathcal{R}} (u \twoheadrightarrow x)$, where $x$ is a fresh variable.

**Definition 6.** *Consider a set of equations $C$, a term $s$ and a fresh variable $x$. We say that a constructor-based conditional narrowing derivation of the form $(C, s \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_\sigma (t \twoheadrightarrow x)$ is* successful *if $t \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ is a constructor term.*

*We say that a derivation of the form $(C, s \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_\sigma (C', s' \twoheadrightarrow x)$ is a* failing *derivation if no more narrowing steps can be applied and at least one of the following following holds:*

- *$C'$ is not the empty sequence, or*
- *$s'$ is not a constructor term.*

*Otherwise, we say that it is an* incomplete *derivation.*

Because of the non-determinism of rule narrowing, the computation of all narrowing derivations starting from a given term is usually represented by means of a narrowing *tree*:

**Definition 7 (constructor-based conditional narrowing tree).** *Let $\mathcal{R}$ be a DCTRS and $C$ be a sequence of equations. A (possibly incomplete)* constructor-based conditional narrowing tree *for $C$ in $\mathcal{R}$ is a (possibly infinite) directed rooted node- and edge-labeled graph $\tau$ built as follows:*

- *the root node of $\tau$ is labeled with $C$;*
- *every node $C_1$ is either a leaf (a node with no output edge) or it is unfolded as follows: there is an output edge from node $C_1$ to node $C_2$ labeled with $\sigma$ for all constructor-based conditional narrowing steps $C_1 \overset{c}{\leadsto}_\sigma C_2$ for the selected innermost narrowable term;*
- *the root node is not a leaf—at least the root node should be unfolded.*

*By abuse of notation, we will denote a finite constructor-based conditional narrowing tree $\tau$ (and its subtrees) for $C$ as a finite set with the constructor-based conditional narrowing derivations starting from $C$ in this tree, i.e., $C \overset{c}{\leadsto}{}^*_\theta C' \in \tau$ if there is a root-to-leaf path from $C$ to $C'$ in $\tau$ labeled with substitutions $\theta_1, \theta_2, \ldots, \theta_n$ such that $C \overset{c}{\leadsto}_{\theta_1} \ldots \overset{c}{\leadsto}_{\theta_n} C'$ is a constructor-based conditional narrowing derivation and $\theta = \theta_n \circ \cdots \circ \theta_1$.*

*Example 4.* Consider again the system $\mathcal{R}$ from Example 2 and the initial sequence of equations $C = (\mathsf{inv}(x_2) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w)$, where $w$ is a fresh variable (the reason to consider this initial sequence of equations will be clear in Definition 9 below). The depth-1 (i.e., derivations are stopped after one narrowing step) constructor-based conditional narrowing tree $\tau$ for $C$ contains the following derivations:

$$
\begin{array}{ll}
C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{left} | x_2']\}} & (\mathsf{inv}(x_2') \twoheadrightarrow (x_1', x'),\ \mathsf{inv}'(x_1') \twoheadrightarrow (t', y'), \\
 & \quad (t', \mathsf{n}(x', y')) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w) \\
C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{str}(x') | t']\}} & ((t', \mathsf{sym}(x')) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w) \\
C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{left}, \mathsf{right} | t']\}} & ((t', \mathsf{bottom}) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w)
\end{array}
$$

Note that a constructor-based conditional narrowing tree can be *incomplete* in the sense that we do not require all unfoldable nodes to be unfolded (i.e., some finite derivations in the tree may be incomplete). Nevertheless, if a node is selected to be unfolded, it should be unfolded in all possible ways using constructor-based conditional narrowing (i.e., one cannot *partially* unfold a node by ignoring some matching rules, which would give rise to incorrect results). In order to keep the tree finite, one can introduce a heuristics that determines when the construction of the tree should terminate. We consider the definition of a particular strategy for ensuring termination out of the scope of this paper; nevertheless, one could use a simple depth-$k$ strategy (as in the previous example) or some more elaborated strategies based on well-founded or well-quasi orderings [10] (as in narrowing-driven partial evaluation [1]).

Let us now recall the notion of *least (general) generalization* [34] (also called *anti-unification* [33]), which will be required to compute a common generalization of all instances of a term by a set of narrowing derivations.

**Definition 8 (least general generalization [34],** *lgg*). *Given two terms $s$ and $t$, we say that $w$ is a generalization of $s$ and $t$ if $w \leqslant s$ and $w \leqslant t$; moreover, it is called the least general generalization of $s$ and $t$, denoted by $lgg(s,t)$, if $w' \leqslant w$ for all other generalizations $w'$ of $s$ and $t$. This notion is extended to sets of terms in the natural way: $lgg(\{t_1,\ldots,t_n\}) = lgg(t_1, lgg(t_2,\ldots lgg(t_{n-1},t_n)\ldots))$ (with $lgg(\{t_1\}) = t_1$ when $n = 1$).*

An algorithm for computing the least general generalization can be found, e.g., in [11]. For instance, $lgg(\mathsf{f}(\mathsf{a},\mathsf{g}(\mathsf{a})),\mathsf{f}(\mathsf{b},\mathsf{g}(\mathsf{b}))) = \mathsf{f}(x_3,\mathsf{g}(x_3))$.

We now introduce a constructive algorithm to produce a more specific version of a rule:

**Definition 9 (**MSV **algorithm for DCTRSs).** *Let $\mathcal{R}$ be a DCTRS and let $l \to r \Leftarrow C \in \mathcal{R}$ be a conditional rewrite rule such that not all terms in $r$ and $C$ are constructor terms. Let $\tau$ be a finite (possibly incomplete) constructor-based conditional narrowing tree for $(C, r \twoheadrightarrow x)$ in $\mathcal{R}$, where $x$ is a fresh variable, and $\tau' \subseteq \tau$ be the tree obtained from $\tau$ by excluding the failing derivations (if any). We compute a more specific version of $l \to r \Leftarrow C$ in $\mathcal{R}$, denoted by* $\mathrm{MSV}(\mathcal{R}, l \to r \Leftarrow C, \tau)$ *, as follows:*

- *If $\tau' = \varnothing$, then* $\mathrm{MSV}(\mathcal{R}, l \to r \Leftarrow C, \tau) = \bot$*, where $\bot$ is used to denote that the rule is useless (i.e., no successful constructor-based reduction can use it).*
- *If $\tau' \neq \varnothing$, then we let $\tau' = \tau_1 \uplus \ldots \uplus \tau_n$ be a partition of the set $\tau$ such that $\tau_i \neq \varnothing$ for all $i = 1,\ldots,n$. Then,*[5]

$$\mathrm{MSV}(\mathcal{R}, l \to r \Leftarrow C, \tau) = \{(l \to r \Leftarrow C)\sigma_1, \ldots, (l \to r \Leftarrow C)\sigma_n\}$$

*where $(l \to r \Leftarrow C)\sigma_i = lgg(\{(l \to r \Leftarrow C)\theta \mid (C, r \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_\theta (C', r' \twoheadrightarrow x) \in \tau_i\})$ with $\mathcal{D}\mathsf{om}(\sigma_i) \subseteq \mathcal{V}\mathsf{ar}(C) \cup \mathcal{V}\mathsf{ar}(r)$, $i = 1,\ldots,n$.*[6]

[5] The *lgg* operator is trivially extended to equations by considering them as terms, e.g., the sequence $s_1 \twoheadrightarrow t_1, s_2 \twoheadrightarrow t_2$ is considered as the term $\wedge(\twoheadrightarrow(s_1,t_1), \twoheadrightarrow(s_1,t_2))$, where $\twoheadrightarrow$ and $\wedge$ are binary function symbols.

[6] By definition of constructor-based conditional narrowing, it is clear that $\sigma_1,\ldots,\sigma_n$ are constructor substitutions.

Regarding the partitioning of the derivations in the constructor-based conditional narrowing tree $\tau'$, i.e., computing $\tau_1, \ldots, \tau_n$ such that $\tau' = \tau_1 \uplus \ldots \uplus \tau_n$, we first apply a simple pre-processing to avoid trivial overlaps between the generated rules: we remove from $\tau'$ those derivations $(C, r \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_\theta (C', r' \twoheadrightarrow x)$ such that there exists another derivation $(C, r \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_{\theta'} (C'', r'' \twoheadrightarrow x)$ with $\theta' \leqslant \theta$. In this way, we avoid the risk of having such derivations in different subtrees, $\tau_i$ and $\tau_j$, thus producing overlapping rules. Once these derivations have been removed, we could proceed as follows:

- No partition ($n = 1$). This is what is done in the original transformation for logic programs [24] and gives good results for most examples (i.e., produces a non-overlapping system).
- Consider a maximal partitioning, i.e., each $\tau_i$ just contains a single derivation. This strategy might produce poor results when the computed substitutions overlap, since overlapping instances would then be produced (even if the considered function was originally non-overlapping).
- Use a heuristics that tries to produce a non-overlapping system whenever possible. Basically, it would proceed as follows. Assume we want to apply the MSV transformation to a function $f$. Let $k$ be a natural number greater than the maximum depth of the left-hand sides of the rules defining $f$. Then, we partition the tree $\tau'$ as $\tau_1, \ldots, \tau_n$ so that it satisfies the following condition (while keeping $n$ as small as possible):
  - for each $(C, r \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_\theta C_1$ and $(C, r \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_{\theta'} C_2$ in $\tau_i$, we have that $top_k(l\theta) = top_k(l\theta')$, where $l \to r \Leftarrow C$ is the considered rule.

  Here, given a fresh constant $\top$, $top_k$ is defined as follows: $top_0(t) = \top$, $top_k(x) = x$ for $k > 0$, $top_k(f(t_1, \ldots, t_m)) = f(top_{k-1}(t_1), \ldots, top_{k-1}(t_m))$ for $k > 0$, i.e., $top_k(t)$ returns the topmost symbols of $t$ up to depth $k$ and replaces the remaining subterms by the fresh constant $\top$.

*Example 5.* Let us apply the MSV algorithm to the first rule of the system $\mathcal{R}$ introduced in Example 2. Here, we consider the depth-1 constructor-based conditional narrowing tree $\tau$ with the derivations shown in Example 4. We first remove derivations labeled with less general substitutions, so that from the first and third derivations of Example 4, only the following one remains:

$$\begin{array}{ll} C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{left}|x_2']\}} & (\mathsf{inv}(x_2') \twoheadrightarrow (x_1', x'),\ \mathsf{inv}'(x_1') \twoheadrightarrow (t', y'), \\ & \quad (t', \mathsf{n}(x', y')) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w) \\ C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{str}(x')|t']\}} & (t', \mathsf{sym}(x')) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w \end{array}$$

Now, either by considering a maximal partitioning or the one based on function $top_k$, we compute the following partitions:

$$\begin{array}{l} \tau_1 = \\ \{C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{left}|x_2']\}} \quad (\mathsf{inv}(x_2') \twoheadrightarrow (x_1', x'),\ \mathsf{inv}'(x_1') \twoheadrightarrow (t', y'), \\ \qquad\qquad (t', \mathsf{n}(x', y')) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w)\} \\ \tau_2 = \\ \{C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{str}(x')|t']\}} ((t', \mathsf{sym}(x')) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w)\} \end{array}$$

so that the computed more specific version, $\mathcal{R}_{msv}$, contains the two rules already shown in Example 2.

Now, we consider the correctness of the MSV transformation.

**Theorem 2.** *Let $\mathcal{R}$ be a DCTRS, $l \to r \Leftarrow C \in \mathcal{R}$ be a rewrite rule such that not all terms in $r$ and $C$ are constructor terms. Let $\tau$ be a finite (possibly incomplete) constructor-based conditional narrowing tree for $(C, r \twoheadrightarrow x)$ in $\mathcal{R}$. Then,*

- *If* $\mathrm{MSV}(\mathcal{R}, l \to r \Leftarrow C, \tau) = \mathcal{R}_{msv}$, *then* $\mathcal{R}_{msv}$ *is a more specific version of* $l \to r \Leftarrow C$ *in* $\mathcal{R}$ *w.r.t.* $\overset{c}{\to}_{\mathcal{R}}$.
- *If* $\mathrm{MSV}(\mathcal{R}, l \to r \Leftarrow C, \tau) = \bot$, *then* $l \to r \Leftarrow C$ *is not used in any successful reduction in* $\mathcal{R}$ *w.r.t.* $\overset{c}{\to}_{\mathcal{R}}$.

The proof can be found in the full version of this paper [31].

## 5 Conclusion and Future Work

We have introduced the notion of a *more specific* version of a rewrite rule in the context of conditional rewrite systems with some restrictions (i.e., typical functional programs). The transformation is useful to produce non-overlapping systems from overlapping ones while preserving the so called successful reductions. We have introduced an automatic algorithm for computing more specific versions and have proved its correctness.

We have undertaken the extension of the implemented program inverter of [28] with a post-process based on the MSV transformation. We have tested the resulting transformation with the 15 program inversion benchmarks of [20].[7] In nine of these benchmarks (out of fifteen) an overlapping system was obtained by inversion while the remaining six are non-overlapping. By applying the MSV transformation to all overlapping rules—except for a predefined operator du—using a depth-3 constructor-based conditional narrowing tree in all examples, we succeeded in improving the nine overlapping systems, while the other six systems were left unchanged: nine overlapping systems are transformed into non-overlapping ones, and the remaining six are transformed into themselves. These promising results point out the usefulness of the approach to improve the quality of inverted systems.

As for future work, we plan to explore the use of the MSV transformation to improve the accuracy of termination analyses.

## References

1. E. Albert and G. Vidal. The narrowing-driven approach to functional logic program specialization. *New Generation Comput.*, 20(1):3–26, 2002.

[7] Unfortunately, the site shown in [20] is not accessible now. Some of the benchmarks can be found in [14, 15, 21]. All the benchmarks are reviewed in [25] and also available from the following URL: `http://www.trs.cm.is.nagoya-u.ac.jp/repius/`. The two benchmarks pack and packbin define non-tail-recursive functions which include some tail-recursive rules; we transform them into pure tail recursive ones using the method introduced in [29] (to avoid producing non-operationally terminating programs), that are also inverted using the approach of [28].

2. J. M. Almendros-Jiménez and G. Vidal. Automatic partial inversion of inductively sequential functions. In *Proc. of the 18th International Symposium on Implementation and Application of Functional Languages*, vol. 4449 of *LNCS*, pp. 253–270, Springer, 2006.
3. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proc. of the 22nd International Conference on Logic Programming*, vol. 4079 of *LNCS*, pp. 87–101, Springer, 2006.
4. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
5. H. Bertling and H. Ganzinger. Completion-time optimization of rewrite-time goal solving. In *Proc. of the 3rd International Conference of Rewriting Techniques and Applications*, vol. 355 of *LNCS*, pp. 45–58, Springer, 1989.
6. A. Bockmayr and A. Werner. LSE narrowing for decreasing conditional term rewrite systems. In *Proc. of the 4th International Workshop on Conditional and Typed Rewriting Systems*, vol. 968 of *Lecture Notes in Computer Science*, pp. 51–70, Springer, 1995.
7. A. Bondorf. Towards a self-applicable partial evaluator for term rewriting systems. In *Proc. of the International Workshop on Partial Evaluation and Mixed Computation*, pp. 27–50. North-Holland, Amsterdam, 1988.
8. A. Bondorf. A self-applicable partial evaluator for term rewriting systems. In *Proc. of International Conference on Theory and Practice of Software Development, Barcelona, Spain*, vol. 352 of *LNCS*, pp. 81–95, Springer, 1989.
9. P. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theor. Comput. Sci.*, 59:3–23, 1988.
10. N. Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1&2):69–115, 1987.
11. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, vol. B: Formal Models and Semantics, pp. 243–320, Elsevier, Amsterdam, 1990.
12. L. Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. of the Symposium on Logic Programming*, pp. 172–185, IEEE Press, 1985.
13. H. Ganzinger. Order-sorted completion: The many-sorted way. *Theor. Comput. Sci.*, 89(1):3–32, 1991.
14. R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In *Proc. of the first Asian Symposium on Programming Languages and Systems*, vol. 2895 of *LNCS*, pp. 246–264, Springer, 2003.
15. R. Glück and M. Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundam. Inform.*, 66(4):367–395, 2005.
16. M. Hanus. The integration of functions into logic programming: From theory to practice. *J. Log. Program.*, 19&20:583–628, 1994.
17. P. G. Harrison. Function inversion. In *Proc. of the International Workshop on Partial Evaluation and Mixed Computation*, pp. 153–166, North-Holland, Amsterdam, 1988.
18. S. Hölldobler. *Foundations of Equational Logic Programming*, vol. 353 of *LNAI*, Springer, 1989.
19. J.-M. Hullot. Canonical forms and unification. In *Proc. of the 5th International Conference on Automated Deduction*, vol. 87 of *LNCS*, pp. 318–334, Springer, 1980.
20. M. Kawabe and Y. Futamura. Case studies with an automatic program inversion system. In *Proc. of the 21st Conference of Japan Society for Software Science and Technology*, number 6C-3, 5 pages, 2004.
21. M. Kawabe and R. Glück. The program inverter LRinv and its structure. In *Proc. of the 7th International Symposium on Practical Aspects of Declarative Languages*, vol. 3350 of *LNCS*, pp. 219–234, Springer, 2005.

22. H. Khoshnevisan and K. M. Sephton. InvX: An automatic function inverter. In *Proc. of the 3rd International Conference of Rewriting Techniques and Applications*, vol. 355 of *LNCS*, pp. 564–568, Springer, 1989.
23. M. Marchiori. On deterministic conditional rewriting. Technical Report MIT-LCS-TM-405, MIT Laboratory for Computer Science, 1997. Available from `http://www.w3.org/People/Massimo/papers/MIT-LCS-TM-405.pdf`.
24. K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990. Available from `http://www.springerlink.com/content/k3p11kl316m73764/`.
25. N. Nishida and M. Sakai. Completion after program inversion of injective functions. *Electr. Notes Theor. Comput. Sci.*, 237:39–56, 2009.
26. N. Nishida, M. Sakai, and T. Sakabe. Generation of inverse computation programs of constructor term rewriting systems. *IEICE Trans. Inf. & Syst.*, J88-D-I(8):1171–1183, 2005 (in Japanese).
27. N. Nishida, M. Sakai, and T. Sakabe. Partial inversion of constructor term rewriting systems. In *Proc. of the 16th International Conference on Rewriting Techniques and Applications*, vol. 3467 of *LNCS*, pp. 264–278, Springer, 2005.
28. N. Nishida and G. Vidal. Program inversion for tail recursive functions. In *Proc. of the 22nd International Conference on Rewriting Techniques and Applications*, vol. 10 of *LIPIcs*, pp. 283–298, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011.
29. N. Nishida and G. Vidal. Conversion to first-order tail recursion for improving program inversion, 2012. *Submitted for publication.*
30. N. Nishida and G. Vidal. More specific term rewriting systems. In *the 21st International Workshop on Functional and (Constraint) Logic Programming*, Nagoya, Japan, 2012. Informal proceedings, available from the URL: `http://www.dsic.upv.es/~gvidal/german/wflp12/paper.pdf`.
31. N. Nishida and G. Vidal. Computing more specific versions of conditional rewriting systems. Technical report, available from the following URL: `http://www.dsic.upv.es/~gvidal/german/lopstr12/paperTR.pdf`.
32. E. Ohlebusch. *Advanced topics in term rewriting.* Springer-Verlag, UK, 2002.
33. F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Proc. of the Sixth Annual Symposium on Logic in Computer Science*, pp. 74–85, IEEE Computer Society, 1991.
34. G. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.
35. J. G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. In *Proc. of the 10th ACM SIGPLAN International Conference on Functional Programming*, pp. 228–239, ACM Press, 2005.
36. J. R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *J. ACM*, 21(4):622–642, 1974.