Program Inversion for Tail Recursive Functions

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

Program inversion is a fundamental problem that has been addressed in many different programming settings and applications. In the context of term rewriting, several methods already exist for computing the inverse of an injective function. These methods, however, usually return non-terminating inverted functions when the considered function is tail recursive. In this paper, we propose a direct and intuitive approach to the inversion of tail recursive functions. Our new technique is able to produce good results even without the use of an additional post-processing of determinization or completion. Moreover, when combined with a traditional approach to program inversion, it constitutes a promising approach to define a general method for program inversion. Our experimental results confirm that the new technique compares well with previous approaches.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases term rewriting, program transformation, termination

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.283

Category Regular Research Paper

1 Introduction

Inverse computation for an n-ary function f is, given an output v of f, the calculation of (all) the possible inputs $v_1, \cdots, v_1$ of f such that $f(v_1, \cdots, v_n) = v$ [27, 28]. To be more precise this is usually called *full* inverse computation, in contrast to *partial* one where some inputs are also provided, i.e., given the output v of f and part of its inputs, say $v_{i_1}, \ldots, v_{i_m}$, the partial inverse computation computes the remaining inputs $v_{j_1}, \ldots, v_{j_k}$ such that $f(v_1, \ldots, v_n) = v$ with $\{v_{i_1}, \ldots, v_{i_m}\} \cup \{v_{j_1}, \ldots, v_{j_k}\} = \{v_1, \ldots, v_n\}$ and $\{v_{i_1}, \ldots, v_{i_m}\} \cap \{v_{j_1}, \ldots, v_{j_k}\} = \emptyset$. Two approaches to inverse computation are distinguished [1]: *inverse interpreters* [4, 1, 32, 15] that perform inverse computation taking the output v, the given inputs (if any), and the definition of f as input, and *inversion compilers* [16, 9, 12, 27, 28, 22, 20, 24, 23, 6, 7, 11, 2] that performs *program inversion*. More precisely, inversion compilers take the definition of f as input and compute the definition of a (possibly partial) inverse function f^{-1}. Note that inverse interpreters can be transformed into inversion compilers by producing inverse functions including an inverse interpreter in the target programming language that is specialized for the function being inverted (similarly to, e.g., [15]). *Semi-inversion* [17, 18, 19] is a more general notion than partial inversion that allows the original output to be partially given.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 283–298

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


Typical applications of (full and partial) program inversion are the automatic development of dual programs: encryption and decryption (e.g., cryptographic encoder $\mathsf{enc}(x,k)$ and decoder $\mathsf{dec}(y,k)$ with a symmetric key k), data compression and decompression (e.g., zip/unzip), data/program translation and re-translation between different programming languages, and so on. Given one-half of dual programs, inversion provides the other program automatically and without bugs, thus guaranteeing high reliability. This is specially important for encryption/decryption and compression/decompression since a bug in these programs may cause serious security problems.

The most popular target of program inversion is the class of injective functions (or functions that are injective w.r.t. the unknown arguments when *partial* inversion is considered). Deterministic definitions are expected as inverses of injective functions since the inverse relation for injective functions is one-to-one. However, this is not ensured by existing methods and in many cases overlapping and/or non-terminating functions are produced instead. Thus, the elimination of non-determinism in inverted functions has recently attracted a lot of interest [6, 7, 11, 2, 21]. Indeed, since all the inversion techniques developed so far are essentially similar, one can say that the difficult part of program inversion for injective functions lies in the elimination of the undesired non-determinism.

In the field of term rewriting, a full-inversion method for constructor term rewriting systems (TRS, for short) has been proposed, and later extended to partial inversion [20, 24, 23]. The compiler (fully or partially) inverts a constructor TRS into a conditional term rewriting system (CTRS, for short) that completely defines inverses of functions defined in the original TRS. The *conditional parts* of rewrite rules in the CTRS can be seen as `let`-structures for declaring variables that are locally used in the rewrite rules.

The method for eliminating non-determinism in [21], a post process for the compilers in [20, 24, 23], consists in applying a restricted variant of *completion* to the systems obtained from the inverse conditional systems by *unraveling* [14, 26]. This method requires the conditional systems to be *operationally terminating* [13] (i.e., any derivation is finite) and outputs computationally-equivalent unconditional systems that are terminating and non-overlapping when the method halts successfully. Although this method is quite restrictive and does not always succeed, it was able to successfully transform all the benchmarks shown in [10, 6, 7, 11] with operationally terminating inverses [21], while it was not applicable to the other benchmarks with non-operationally-terminating inverses (see Example 3.2 below). On the other hand, the method for eliminating non-determinism in [6, 7, 11] is based on applying LR parsing techniques to a grammar-based representation of functional programs. This is quite an interesting non-standard application of LR parsing and performs surprisingly well for the benchmarks of [10] that contain several kinds of schemes of function definitions (such as tail-recursion, non-tail-recursion, and the combination of both), despite the fact that only LR(0) parsing is considered. The authors do not consider partial inversion since the grammar-based representation is not adequate to identify known and unknown arguments separately. Moreover, the grammar-based programs cannot express functions containing erasing rules (though these rules arise quite naturally when considering partially inverted programs).

Given an n-ary function f, traditional approaches to (partial) inversion are based on the property "$f(v_1,\cdots,v_n)=v$ iff $f^{-1}(v,v_{i_1},\cdots,v_{i_m})=(v_{j_1},\cdots,v_{j_k})$ where $\{i_1,\cdots,i_m\}$ are known input arguments and $\{i_1,\cdots,i_m\}\uplus\{j_1,\cdots,j_k\}=\{1,\cdots,n\}$", i.e., the equation $f(v_1,\cdots,v_n)=v$ is replaced by $f^{-1}(v,v_{i_1},\cdots,v_{i_m})=(v_{j_1},\cdots,v_{j_k})$ and instruction sequences are inverted [27, 28, 22, 20, 24, 23, 6, 7, 11, 2]. For example, when considering full inversion, a rewrite rule is normalized to a conditional rule $f(t_1,\cdots,t_n)\to t \Leftarrow$

$f_1(\overrightarrow{u_1}) \twoheadrightarrow x_1; \cdots ; f_k(\overrightarrow{u_k}) \twoheadrightarrow x_k$ (see Definition 3.4), and it is inverted to a conditional rule $f^{-1}(t) \to (t_1, \cdots, t_n) \Leftarrow f_k^{-1}(x_k) \twoheadrightarrow (\overrightarrow{u_k}); \cdots ; f_1^{-1}(x_1) \twoheadrightarrow (\overrightarrow{u_1})$ where $t_1, \cdots, t_n, t$ are constructor terms, $\overrightarrow{u_1}, \cdots, \overrightarrow{u_k}$ are sequences of constructor terms, and $x_1, \cdots, x_n$ are variables (see Example 3.1 below). This approach is in principle applicable to arbitrary functions. Unfortunately, for a tail-recursive function defined by a rule like $f(t_1, \cdots, t_n) \to f(u_1, \cdots, u_n)$, this approach generates a non-operationally-terminating rule of the form $f^{-1}(x) \to (t_1, \cdots, t_n) \Leftarrow f^{-1}(x) \twoheadrightarrow (u'_1, \cdots, u'_n); \cdots$ when full inversion is considered.

When the first argument of the inverse f^{-1} takes as input the output of the original function f (i.e., the input value of the first argument of f^{-1} is in the range of f), a breadth-first search is enough to get the output (i.e., the original input) since there exists a finite path from the original input to the original output. Therefore, for a non-operationally-terminating inverse system of an injective function, a breadth-first search is enough to compute the output. However, the finiteness of the breadth-first search is guaranteed only when the input is one of the outputs of the original function. Thus, in general, the breadth-first search might be non-terminating (i.e., the search space might be infinite). Furthermore, when the given function is not surjective (which is itself difficult to know), it is not easy to determine whether the input is one of the outputs of the original function or not. Moreover, practical rewriting systems (or functional programming environments) do not usually implement breadth-first search strategies. For these reasons, the (operational) termination of inverted systems is desired.

As stated above, the non-determinism elimination method of [6, 7, 11] can solve the non-determinism of some inverted programs so that the resulting system is terminating. This method, however, does not succeed for all inverted systems. On the other hand, the method in [21] cannot be applied to any non-operationally-terminating system since the method requires termination. As mentioned before, when full inversion is considered, tail recursive rules of the form $f(t_1, \cdots, t_n) \to f(u_1, \cdots, u_n)$ are inverted to $f^{-1}(x) \to (t_1, \cdots, t_n) \Leftarrow f^{-1}(x) \twoheadrightarrow (u'_1, \cdots, u'_n); \cdots$ that cause non-termination (since $f^{-1}(x)$ calls $f^{-1}(x)$ again). Therefore, the traditional approaches to inversion are not suitable for tail recursive rules (unless some post-processing is applied to recover more suitable definitions). Actually, the basic methods for program inversion have not been significantly improved since their original definitions (the focus has been in the development of methods for eliminating non-determinism instead). However, we think that there is still room for improving the current inversion techniques.

In this paper, we introduce a novel approach to program inversion which is specially tailored to deal with tail recursive functions defined by means of a rewrite system. We combine this approach with the previous technique in [20, 23] to produce a general inversion method. For the sake of readability, we only consider the full inversion of functions (as in [20, 23]), though it would not be difficult to combine our technique with the approach to partial inversion of [20, 24]. In addition, we do not consider *sorts* in this paper, though the results can be straightforwardly extended to many-sorted systems. Our research is motivated by the fact that tail recursive functions are extensively used due to their good computational properties (i.e., they are usually compiled as iterations, which are much more efficient than standard recursive functions).

This paper is organized as follows. In Section 2 we briefly review notions and notations of term rewriting. Section 3 introduces an inversion method for tail recursive functions. Section 4 shows a summary of our experimental evaluation and compare the new approach with related works. Finally, Section 5 concludes and points out some directions for future research. Proofs of technical results can be found in the extended version [25] of this paper.

2 Preliminaries

In this section, we recall some basic notions and notations of term rewriting [3, 26].

Throughout this paper, we use $\mathcal{V}$ as a countably infinite set of *variables*. Let $\mathcal{F}$ be a *signature*, i.e., a finite set of *function symbols* with a fixed arity denoted by $\mathsf{ar}(f)$ for a function symbol f. The set of *terms* over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F},\mathcal{V})$, and the set of variables appearing in terms $t_1,\cdots,t_n$ is denoted by $\mathcal{V}ar(t_1,\cdots,t_n)$. The *identity* of terms s and t is written by $s \equiv t$. The notation $C[t_1,\cdots,t_n]_{p_1,\cdots,p_n}$ represents the term obtained by replacing each *hole* $\Box$ at *position* p_i of an n-hole *context* $C[\ ]$ with term t_i for $1 \leq i \leq n$. We may omit the subscripts $p_1,\cdots,p_n$ when they are clear from the context. The *domain* and *range* of a *substitution* σ are denoted by $\mathcal{D}om(\sigma)$ and $\mathcal{R}an(\sigma)$, respectively; a substitution σ will be denoted by $\{x_1 \mapsto t_1,\cdots,x_n \mapsto t_n\}$ if $\mathcal{D}om(\sigma) = \{x_1,\cdots,x_n\}$ and $\sigma(x_i) \equiv t_i$ for $1 \leq i \leq n$. The set of variables appearing in the range of σ is denoted by $\mathcal{VR}an(\sigma)$: $\mathcal{VR}an(\sigma) = \bigcup_{x\in\mathcal{D}om(\sigma)} \mathcal{V}ar(x\sigma)$. The application $\sigma(t)$ of substitution σ to term t is abbreviated to $t\sigma$.

An (*oriented*) *conditional rewrite rule* over a signature $\mathcal{F}$ is a triple (l,r,c), denoted by $l \to r \Leftarrow c$, such that the *left-hand side* l is a non-variable term of $\mathcal{T}(\mathcal{F},\mathcal{V})$, the *right-hand side* r is a term of $\mathcal{T}(\mathcal{F},\mathcal{V})$, and the *conditional part* c is a sequence $s_1 \twoheadrightarrow t_1;\cdots;s_k \twoheadrightarrow t_k$ $(k \geq 0)$ where $s_1,\cdots,s_k,t_1,\cdots,t_k$ are terms of $\mathcal{T}(\mathcal{F},\mathcal{V})$. In particular, the rewrite rule is called *unconditional* if the conditional part is the empty sequence (i.e., $k = 0$), and we may abbreviate it to $l \to r$. We sometimes attach a unique label ρ to rule $l \to r \Leftarrow c$, written $\rho : l \to r \Leftarrow c$, so that we can use the label to refer to this rule. The set of variables in c and ρ are denoted by $\mathcal{V}ar(c)$ and $\mathcal{V}ar(\rho)$, resp.: $\mathcal{V}ar(s_1 \twoheadrightarrow t_1;\cdots;s_k \twoheadrightarrow t_k) = \mathcal{V}ar(s_1,t_1,\cdots,s_k,t_k)$ and $\mathcal{V}ar(\rho) = \mathcal{V}ar(l,r,c)$.

An (*oriented*) *conditional term rewriting system* (*CTRS*, for short) R over a signature $\mathcal{F}$ is a finite set of conditional rewrite rules over $\mathcal{F}$. In particular, R is called an (*unconditional*) *term rewriting system* (*TRS*, for short) if every rule $l \to r \Leftarrow c$ in R is unconditional and satisfies $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. The *rewrite relation* $\to_R$ of R is defined as follows: $\to_{(0),R} = \emptyset$, $\to_{(i+1),R} = \to_{(i),R} \cup\{(C[l\sigma],C[r\sigma]) \mid l \to r \Leftarrow s_1 \twoheadrightarrow t_1;\cdots;s_k \twoheadrightarrow t_k \in R,\ s_1\sigma \to^*_{(i),R} t_1\sigma,\ \cdots,\ s_k\sigma \to^*_{(i),R} t_k\sigma\}$ for $i \geq 0$, and $\to_R = \bigcup_{i\geq 0} \to_{(i),R}$. A notion of *operational termination* of CTRSs is defined via the absence of infinite well-formed proof trees in some inference system [13]: a CTRS R is *operationally terminating* if for any terms s and t, any proof tree attempting to prove that $s \to^*_R t$ cannot be infinite.

A conditional rewrite rule $l \to r \Leftarrow s_1 \twoheadrightarrow t_1;\cdots;s_k \twoheadrightarrow t_k$ is called *deterministic* if $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(l,t_1,\cdots,t_{i-1})$ for $1 \leq i \leq k$. Note that the terminology "deterministic" refers to the sequential evaluation of conditional parts, i.e., when a CTRS is deterministic, the conditional parts can be evaluated from left to right. In contrast, the determinacy of the rewrite rule application is denoted with the terminology "non-overlapping". A CTRS is called *deterministic* (*DCTRS*, for short) if all of its rules are deterministic. A CTRS is called *non-erasing* if every rule $l \to r \Leftarrow c$ satisfies $\mathcal{V}ar(l) \subseteq \mathcal{V}ar(r)$.

Let R be a CTRS over a signature $\mathcal{F}$. The set of *defined symbols of* R is denoted by $\mathcal{D}_R = \{\mathsf{root}(l) \mid l \to r \Leftarrow c \in R\}$ and the set of *constructors of* R is denoted by $\mathcal{C}_R = \mathcal{F} \setminus \mathcal{D}_R$. A term in $\mathcal{T}(\mathcal{C}_R,\mathcal{V})$ is called a *constructor term of* R. The CTRS R is called a *constructor system* if for each rule in R, any proper subterm of the left-hand side is a constructor term of R. A substitution σ is called a *constructor substitution of* R if $\mathcal{R}an(\sigma) \subseteq \mathcal{T}(\mathcal{C}_R,\mathcal{V})$. For a term t, $\mathsf{cap}(t)$ is a term obtained by replacing each proper subterm rooted by a defined symbol with a fresh variable [26].

As a model of the call-by-value evaluation, we define the *constructor-based reduction*

relation $\xrightarrow[c]{}_R$ of a CTRS R, a restricted variant of *constructor rewriting* in [31], as follows:

- $\xrightarrow[c]{}_{(0),R} = \emptyset$,
- $\xrightarrow[c]{}_{(n+1),R} = \xrightarrow[c]{}_{(n),R} \cup \{(C[l\theta], C[r\theta]) \mid l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots; s_k \twoheadrightarrow t_k \in R,\ \mathcal{R}an(\theta) \subseteq \mathcal{T}(\mathcal{C}_R, \mathcal{V}),\ s_1\theta \xrightarrow[c]{}^*_{(n),R} t_1\theta,\ \cdots,\ s_k\theta \xrightarrow[c]{}^*_{(n),R} t_k\theta\}$, where $n \geq 0$, and
- $\xrightarrow[c]{}_R = \bigcup_{i>0} \xrightarrow[c]{}_{(i),R}$.

Roughly speaking, for a CTRS that can be regarded as a functional program, the constructor-based reduction corresponds to the *call-by-value* evaluation where constructor terms are considered as data objects. Note that $\xrightarrow[c]{}_R$ is a strict subrelation of the *operationally innermost reduction* [21], the innermost reduction of DCTRSs, if every rule $l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots; s_k \twoheadrightarrow t_k$ satisfies that $t_1, \cdots, t_k \in \mathcal{T}(\mathcal{C}_R, \mathcal{V})$. Unlike the innermost reduction of CTRSs (cf. [8]), $\xrightarrow[c]{}_R$ is well-defined for every CTRS since constructor terms are well-defined while normal forms of non-operationally-terminating CTRSs are not well-defined.

Let R be a CTRS. Two rewrite rules $l_1 \to r_1 \Leftarrow c_1$ and $l_2 \to r_2 \Leftarrow c_2$ are called *overlapping* if there exists a context $C[\,]$ and a non-variable term t such that $l_2 \equiv C[t]$ and l_1 and t are unifiable, where we assume w.l.o.g. that these rules share no variable. Then, a conditional pair of terms $((C[r_1])\theta, r_2\theta) \Leftarrow c_1\theta; c_2\theta$ is called a *critical pair* of R where θ is a most general unifier of l_1 and t. A critical pair $(s, t) \Leftarrow c$ is called *trivial* if $s \equiv t$, and it is called *infeasible w.r.t.* $\xrightarrow[c]{}_R$ if for any substitution σ, c contains a condition $u \twoheadrightarrow v$ such that $u\sigma \not\xrightarrow[c]{}^*_R v\sigma$ [33].

Let R be a DCTRS and f be a defined symbol of R. A rule $l \to r$ in R is called an *f-rule* if $\mathsf{root}(l)$ is f. We denote the set of f-rules in R by $R|_f$. For a set D of defined symbols, $R|_D$ denotes $\bigcup_{g \in D} R|_g$. The set $\mathsf{Dep}_R(f)$ of defined symbols that f *depends on* is the minimum set such that

- $\mathsf{Dep}_R(f)$ contains every defined symbol appearing in r, c of $l \to r \Leftarrow c \in R|_f$, and
- $\mathsf{Dep}_R(f) \supseteq \mathsf{Dep}_R(g)$ if $g \in \mathsf{Dep}_R(f)$.

The symbol f is called (*mutually*) *recursive* if there exists a symbol g in $\mathsf{Dep}_R(f)$ that depends on f. Moreover, f is called *self-recursive* if the only such symbol is f itself. An f-rule $l \to r$ is called a *non-recursive rule* if any defined symbol in r does not depend on f; otherwise, the f-rule is called a *recursive rule*. Let $t_1, \cdots, t_n$ be constructor terms of R. Then the term $f(t_1, \cdots, t_n)$ is called a *call pattern*.

We introduce special constructors $\mathsf{tp}_0, \mathsf{tp}_1, \mathsf{tp}_2, \cdots$ with $\mathsf{ar}(\mathsf{tp}_i) = i$ in order to denote tuples (records) of terms, e.g., the tuple $(t_1, \cdots, t_n)$ of terms $t_1, \cdots, t_n$ is denoted by $\mathsf{tp}_n(t_1, \cdots, t_n)$. The reason why these constructors are introduced is that the inverses of n-ary functions with $n > 1$ will return tuples of terms. A CTRS R is called tp*-free* if it does not contain any tuple symbol. Although it may seem trivially correct in a functional setting, we will never replace $\mathsf{tp}_1(t)$ by t in our term rewriting context. The reason is that, given a CTRS R that is operationally terminating w.r.t. $\to_R$, the CTRS obtained from R by replacing each occurrence of $\mathsf{tp}_1(t)$ by t is not always operationally terminating w.r.t. $\to_R$ while it is operationally terminating w.r.t. $\xrightarrow[c]{}_R$.

We are now ready to introduce our notion of full inverses for functions:

▶ **Definition 2.1** (full inverse)**.** Let R be a tp-free CTRS over a signature $\mathcal{F}$ and S be a CTRS over a signature $\mathcal{G}$ such that $\mathcal{C}_R \subseteq \mathcal{C}_S$. A defined symbol g of S is called a *full inverse* of f if

- for any constructor terms $t_1, \cdots, t_n, u$ of R, $f(t_1, \cdots, t_n) \xrightarrow[c]{}^*_R u$ if and only if $g(u) \xrightarrow[c]{}^*_S \mathsf{tp}_n(t_1, \cdots, t_n)$.

In the following, the full inverse of a function f is denoted by f^{-1}. We say that a rewrite system S is a *full inverse system of f in R* if S defines f^{-1}.

3 Inversion Transformation for Tail Recursive Functions

In this section, we introduce a method for the inversion of constructor TRSs which extends the previous approach of [20, 23] with an appropriate technique for the inversion of tail recursive functions.

▶ **Example 3.1.** Consider the following rewrite system defining the well-known reverse function with an accumulating parameter:

$$R_{\mathsf{reverse}} = \left\{ \begin{array}{c} \mathsf{reverse}(xs) \to \mathsf{rev}(xs, \mathsf{nil}) \\ \mathsf{rev}(\mathsf{nil}, ws) \to ws \\ \mathsf{rev}(\mathsf{cons}(x, xs), ys) \to \mathsf{rev}(xs, \mathsf{cons}(x, ys)) \end{array} \right\}$$

This system is inverted by the previous inversion technique [20, 23, 6, 7, 11] as follows:

$$\overline{R_{\mathsf{reverse}}} = \left\{ \begin{array}{l} \mathsf{reverse}^{-1}(ys) \to \mathsf{tp}_1(xs) \Leftarrow \mathsf{rev}^{-1}(ys) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil}) \\ \quad \mathsf{rev}^{-1}(ws) \to \mathsf{tp}_2(\mathsf{nil}, ws) \\ \quad \mathsf{rev}^{-1}(zs) \to \mathsf{tp}_2(\mathsf{cons}(x, xs), ys) \Leftarrow \mathsf{rev}^{-1}(zs) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{cons}(x, ys)) \end{array} \right\}$$

The functions $\mathsf{reverse}$ and $\mathsf{reverse}^{-1}$ are full inverses of each other, and the functions rev and rev^{-1} are full inverses of each other too. However, $\mathsf{reverse}$ is injective but rev is not. Unfortunately, this inverse system is neither non-overlapping nor operationally terminating. Moreover, the method in [21] is not applicable because the inverted system is not operationally terminating. Luckily, for this example, the non-determinism elimination method in [6, 7, 11] can transform $\overline{R_{\mathsf{reverse}}}$ into a computationally equivalent CTRS that is operationally terminating and non-overlapping. However, this method is not always successful (see Example 3.14 below) and thus we plan to introduce instead a direct approach that is able to produce the right inverse without the need of a post-process (which nevertheless will be still applicable when the direct approach does not succeed).

For the sake of readability, we only consider functions defined by *unconditional* and non-erasing constructor TRSs as a target of inversion. We also require *tail recursive* functions to be *self-recursive*[1] according to the following scheme:

- there is a single non-recursive rule $f(w_1, \cdots, w_n) \to r$ such that the right-hand side r is a constructor term, and
- there are one or more recursive rules of the form $f(t_1, \cdots, t_n) \to f(u_1, \cdots, u_n)$.

These restrictions are not essential and thus the results shown later can easily be extended to functions that do not fulfill them. Given a tail recursive function f, a call pattern $f(t_1, \cdots, t_m)$ is called *initial* if it is a renamed variant of $\mathsf{cap}(t)$ where t is either

- a subterm of the right-hand side of some rule which is not an f-rule, or
- a strict subterm of the right-hand side of some recursive f-rule.

Intuitively speaking, initial call patterns represent initial calls to tail recursive functions.

▶ **Example 3.2.** Consider again the TRS R_{reverse} of Example 3.1. Then, the only initial call of rev in R_{reverse} is $\mathsf{rev}(xs, \mathsf{nil})$.

The next definition introduces a notion of inverse which is specially tailored to tail recursive functions.

[1] It is possible in general to transform mutually recursive functions to a computationally equivalent self-recursive functions, by adding an extra argument to each function to show the original function and by replacing all the recursive function symbols with the same fresh one. However, this transformation is not necessary since the inversion shown later can easily be extended to tail mutually-recursive functions.

▶ **Definition 3.3** (tail recursive inverse)**.** Let R be a tp-free CTRS over a signature $\mathcal{F}$ and S be a CTRS over a signature $\mathcal{G}$ such that $\mathcal{C}_R \subseteq \mathcal{C}_S$. Given a tail recursive function f of $\mathcal{R}$, a defined symbol g of S is called a *tail recursive inverse of* f if

- for any constructor terms $t_1, \cdots, t_n$ of R, $f(t_1, \cdots, t_n) \xrightarrow[c]{}{}^*_R u$ if and only if $u \equiv r\sigma$ and $g(w_1, \cdots, w_n)\sigma \xrightarrow[c]{}{}^*_S \mathsf{tp}_n(t_1, \cdots, t_n)$ for a constructor substitution σ of R and a non-recursive f-rule $f(w_1, \cdots, w_n) \to r$.

In the following, we denote by $\widetilde{f}$ the tail recursive inverse of f.

Let us first illustrate our technique with an example. In Example 3.1 above, one can observe that the computation of reverse has the following scheme:

$$\begin{aligned}
&\mathsf{reverse}(xs)\theta_0 \to \\
&\mathsf{rev}(xs, \mathsf{nil})\theta_0 \equiv \mathsf{rev}(\mathsf{cons}(x_1, xs_1), ys_1)\theta_1 &&\to \mathsf{rev}(xs_1, \mathsf{cons}(x_1, ys_1))\theta_1 \\
&\qquad\equiv \mathsf{rev}(\mathsf{cons}(x_2, xs_2), ys_2)\theta_2 &&\to \mathsf{rev}(xs_2, \mathsf{cons}(x_2, ys_2))\theta_2 \\
&\qquad\qquad\vdots \\
&\qquad\equiv \mathsf{rev}(\mathsf{cons}(x_n, xs_n), ys_n)\theta_n \to \mathsf{rev}(xs_n, \mathsf{cons}(x_n, ys_n))\theta_n &&\equiv \mathsf{rev}(\mathsf{nil}, ws)\theta \to ws\theta
\end{aligned}$$

Therefore, the inverse function $\mathsf{reverse}^{-1}$ should start with a call to $\mathsf{rev}(\mathsf{nil}, ws)\theta$ and should end with a call to $\mathsf{rev}(xs, \mathsf{nil})\theta_0$. Then, the computation of $\widetilde{\mathsf{rev}}$ should follow the pattern

$$\begin{aligned}
\widetilde{\mathsf{rev}}(\mathsf{nil}, ws)\theta \equiv \widetilde{\mathsf{rev}}(xs_n, \mathsf{cons}(x_n, ys_n))\theta_n &\to \widetilde{\mathsf{rev}}(\mathsf{cons}(x_n, xs_n), ys_n)\theta_n \\
&\vdots \\
\equiv \widetilde{\mathsf{rev}}(xs_1, \mathsf{cons}(x_1, ys_1))\theta_1 &\to \widetilde{\mathsf{rev}}(\mathsf{cons}(x_1, xs_1), ys_1)\theta_1 \equiv \widetilde{\mathsf{rev}}(xs, \mathsf{nil})\theta_0
\end{aligned}$$

so that it outputs $\mathsf{tp}_2(xs, \mathsf{nil})\theta_0$. Thus, this pattern means that we should require $\widetilde{\mathsf{rev}}(\mathsf{nil}, ws)$ to be reduced to $\mathsf{tp}_2(xs, \mathsf{nil})$. To summarize, the inversion of the tail-recursive function rev will proceed as follows:

- First, we *normalize* the right-hand side of the rules in order to avoid defined function symbols (except for the topmost one when the function is tail-recursive; see the formal definition below):

$$\left\{\begin{array}{r} \mathsf{reverse}(xs) \to ys \Leftarrow \mathsf{rev}(xs, \mathsf{nil}) \twoheadrightarrow ys \\ \mathsf{rev}(\mathsf{nil}, ws) \to ws \\ \mathsf{rev}(\mathsf{cons}(x, xs), ys) \to \mathsf{rev}(xs, \mathsf{cons}(x, ys)) \end{array}\right\}$$

- Then, the first rule is transformed to

  $$\mathsf{reverse}(xs) \to ys \Leftarrow \mathsf{rev}(xs, \mathsf{nil}) \twoheadrightarrow ws;\ ws \twoheadrightarrow ys$$

  The condition $ws \twoheadrightarrow ys$ is added to make it explicit that ys should be equal to ws, the output of function rev (i.e., the right-hand side of the base case). Now, we basically proceed to exchange the left- and right-hand sides of every equation except for the call to the tail-recursive function rev which is transformed as explained above:

  $$\mathsf{reverse}^{-1}(ys) \to xs \Leftarrow ys \twoheadrightarrow ws;\ \widetilde{\mathsf{rev}}(\mathsf{nil}, ws) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil})$$

- The intermediate reduction steps in the computation for $\widetilde{\mathsf{rev}}$ are done using the inverse of the recursive rule for rev (we just exchanged the left- and right-hand sides):

  $$\widetilde{\mathsf{rev}}(xs, \mathsf{cons}(x, ys)) \to \widetilde{\mathsf{rev}}(\mathsf{cons}(x, xs), ys)$$

- Finally, the base case for $\widetilde{\mathsf{rev}}$ has now the form

  $$\widetilde{\mathsf{rev}}(xs, \mathsf{nil}) \to \mathsf{tp}_2(xs, \mathsf{nil})$$

according to the reasoning above. Therefore, we get the following non-overlapping and operationally terminating system:

$$\overline{R_{\mathsf{reverse}}} = \left\{ \begin{array}{c} \mathsf{reverse}^{-1}(ws) \to xs \Leftarrow \widetilde{\mathsf{rev}}(\mathsf{nil}, ws) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil}) \\ \widetilde{\mathsf{rev}}(xs, \mathsf{nil}) \to \mathsf{tp}_2(xs, \mathsf{nil}) \\ \widetilde{\mathsf{rev}}(xs, \mathsf{cons}(x, ys)) \to \widetilde{\mathsf{rev}}(\mathsf{cons}(x, xs), ys) \end{array} \right\}$$

where the condition $ys \twoheadrightarrow ws$ has been removed by substituting ws with ys.
The idea above is similar to that in [18]. However, there is no formalization nor correctness proofs in [18]. In the rest of this section, we formalize this idea and prove its correctness.

In the previous example, no nested function calls occurred in the right-hand sides of rev. However, we could easily extend the above approach to tail recursive functions whose right-hand sides have nested function calls by dealing with them inductively. Note that we keep applying the old approach of [20, 23] to non-tail-recursive functions.

The following definition introduces a pre-processing of normalization which is similar to that in [2] for functional programs with `let` expressions. The main difference is that the root symbols of the right-hand sides of tail recursive rules are not normalized in our case.

▶ **Definition 3.4** (normalization)**.** Let R be a constructor TRS and f be a defined symbol of R. For an f-rule $l \to r$, the normalized rewrite rule $\mathcal{N}(l \to r)$ is a conditional rewrite rule obtained by repeatedly applying the following transformation until the right-hand side has either no defined symbol or just one at the root position when f is tail recursive:

- Given a rule $l \to C[t] \Leftarrow c$ such that t is rooted by a defined symbol of R and has no defined symbol in its proper subterms, we transform it into $l \to C[x] \Leftarrow c;\ t \twoheadrightarrow x$, where x is a fresh variable.

The normalization $\mathcal{N}$ is extended to TRSs as follows $\mathcal{N}(R) = \{\mathcal{N}(l \to r) \mid l \to r \in R\}$.

The correctness of the normalization process is a consequence of the following results:

▶ **Lemma 3.5.** *Let R and S be constructor CTRSs such that $R = R_0 \uplus \{\rho : l \to C[r] \Leftarrow c\}$ and $S = R_0 \uplus \{\rho' : l \to C[x] \Leftarrow c; r \twoheadrightarrow x\}$ such that x is a fresh variable, r contains a defined symbol of R and, for each $s' \twoheadrightarrow t'$ in c, t' is a constructor term of R. Then, for all terms s and constructor terms t of R, $s \xrightarrow[\mathsf{c}]{}{}^*_R t$ iff $s \xrightarrow[\mathsf{c}]{}{}^*_S t$.*

Proof (Sketch). The *if* and *only-if* parts can be proved by induction on the lexicographic product (k, n) of $\xrightarrow[\mathsf{c}]{}{}^n_{(k),S}$ and $\xrightarrow[\mathsf{c}]{}{}^n_{(k),R}$, respectively (see [25]). ◀

▶ **Lemma 3.6.** *Let R be a constructor TRS, s be a term, and t be a constructor term of R. Then, $s \xrightarrow[\mathsf{c}]{}{}^*_R t$ iff $s \xrightarrow[\mathsf{c}]{}{}^*_{\mathcal{N}(R)} t$.*

Proof (Sketch). This lemma is a direct consequence of Lemma 3.5. ◀

For the sake of readability, we assume w.l.o.g. that a constructor TRS contains only the rewrite rules usable for the computation of the main function it defines. Thus, the analysis used in [20, 24, 23] to collect which functions are inverted is not necessary and we simply invert all the functions of the TRS. Moreover, we assume that the main function is not tail recursive since the new approach for tail recursive functions requires at least one initial call to the tail recursive function (as it happens in practice).

▶ **Definition 3.7** (inversion)**.** Let R be a constructor TRS such that its main function is not tail recursive. Then, the transformation $\mathcal{I}nv$ is defined as follows:

TRS inversion $\quad \mathcal{I}nv(R) = \bigcup_{l \to r \Leftarrow c \in \mathcal{N}(R)} \mathcal{I}nv_{\mathsf{rule}}(l \to r \Leftarrow c)$

Rule inversion $\mathcal{I}nv_{\mathsf{rule}}$:

(i) If f is not tail recursive, then

$$\begin{aligned}&\mathcal{I}nv_{\mathsf{rule}}(\ f(u_1,\cdots,u_n)\to r \Leftarrow s_1 \twoheadrightarrow t_1;\cdots;s_k\twoheadrightarrow t_k\ )\\&\quad=\{\ f^{-1}(r)\to \mathsf{tp}_n(u_1,\cdots,u_n)\Leftarrow \mathcal{I}nv_{\mathsf{c}}(s_k\twoheadrightarrow t_k);\cdots;\mathcal{I}nv_{\mathsf{c}}(s_1\twoheadrightarrow t_1)\ \}\end{aligned}$$

(ii) If f is tail recursive and the corresponding rule is a non-recursive f-rule, then

$$\begin{aligned}&\mathcal{I}nv_{\mathsf{rule}}(\ f(u_1,\cdots,u_n)\to r\ )=\\&\quad\{\ \widetilde{f}(w_1,\cdots,w_n)\to \mathsf{tp}_n(w_1,\cdots,w_n)\mid f(w_1,\cdots,w_n)\text{ is an initial call of }f\text{ in }R\ \}\end{aligned}$$

(iii) Otherwise (i.e., if f is tail recursive and the corresponding rule is a recursive f-rule),

$$\begin{aligned}&\mathcal{I}nv_{\mathsf{rule}}(\ f(u_1,\cdots,u_n)\to f(r_1,\cdots,r_n)\Leftarrow s_1\twoheadrightarrow t_1;\cdots;s_k\twoheadrightarrow t_k\ )\\&\quad=\{\ \widetilde{f}(r_1,\cdots,r_n)\to\widetilde{f}(u_1,\cdots,u_n)\Leftarrow\mathcal{I}nv_{\mathsf{c}}(s_k\twoheadrightarrow t_k);\cdots;\mathcal{I}nv_{\mathsf{c}}(s_1\twoheadrightarrow t_1)\ \}\end{aligned}$$

Condition inversion $\mathcal{I}nv_{\mathsf{c}}$:

(i) If f is not tail recursive, then

$$\mathcal{I}nv_{\mathsf{c}}(f(u_1,\cdots,u_n)\twoheadrightarrow x)=f^{-1}(x)\twoheadrightarrow \mathsf{tp}_n(u_1,\cdots,u_n)$$

(ii) Otherwise,

$$\mathcal{I}nv_{\mathsf{c}}(f(u_1,\cdots,u_n)\twoheadrightarrow x)=x\twoheadrightarrow r;\widetilde{f}(v_1,\cdots,v_n)\twoheadrightarrow \mathsf{tp}_n(u_1,\cdots,u_n)$$

where $f(v_1,\cdots,v_n)\to r$ is a renamed variant of a non-recursive f-rule such that the variables in $v_1,\cdots,v_n$ are fresh.

Each inversion according to Definition 3.7 proceeds as follows:

- Rule inversion (i) and Condition inversion (i) process non-tail-recursive functions similarly to the previous inversion methods of [20, 23].
- Rule inversion (ii) generates a non-recursive rule of $\widetilde{f}$ for a tail recursive function f. In this case, the input rule $f(u_1,\cdots,u_n)\to r$ is not used, except for the root defined symbol f of the left-hand side. This case is only a trigger to generate non-recursive rules for $\widetilde{f}$, while the discarded terms $u_1,\cdots,u_n,r$ are consumed via $f(v_1,\cdots,v_n)\to r$ in Condition inversion (ii).
- Rule inversion (iii), the main part of our new approach, inverts a recursive rule of a tail recursive function f following the idea described above.
- Condition inversion (ii) inverts the condition of a tail recursive function f into the condition that is an initial call of $\widetilde{f}$, adding a condition that connects x and r.

We will show later how to remove the assumption that non-recursive rules of tail recursive functions are unique (see Example 3.13 below). Returned tuples of $\widetilde{f}$ can sometimes be further optimized (e.g., by eliminating trivial elements such as nil in $\mathsf{tp}_2(xs,\mathsf{nil})$ returned by $\widetilde{\mathsf{rev}}$). However, when there are several initial calls, such optimizations might destroy the correctness of the transformation and should be carefully considered.

▶ **Example 3.8.** Consider the TRS R_{reverse} from Example 3.1 again. Normalization only changes the first rule as follows:

$$\mathcal{N}(\mathsf{reverse}(xs)\to\mathsf{rev}(xs,\mathsf{nil}))=\mathsf{reverse}(xs)\to ys\Leftarrow\mathsf{rev}(xs,\mathsf{nil})\twoheadrightarrow ys$$

Then, each rule in $\mathcal{N}(R_{\mathsf{reverse}})$ is inverted as follows:

$$\begin{aligned}&\mathcal{I}nv_{\mathsf{rule}}(\mathsf{reverse}(xs)\to ys\Leftarrow\mathsf{rev}(xs,\mathsf{nil})\twoheadrightarrow ys)\\&\qquad=\{\ \mathsf{reverse}^{-1}(ys)\to\mathsf{tp}_1(xs)\Leftarrow\mathcal{I}nv_{\mathsf{c}}(\mathsf{rev}(xs,\mathsf{nil})\twoheadrightarrow ys)\ \}\\&\qquad=\{\ \mathsf{reverse}^{-1}(ys)\to\mathsf{tp}_1(xs)\Leftarrow ys\twoheadrightarrow zs;\ \widetilde{\mathsf{rev}}(\mathsf{nil},zs)\twoheadrightarrow\mathsf{tp}_2(xs,\mathsf{nil})\ \}\\&\mathcal{I}nv_{\mathsf{rule}}(\mathsf{rev}(\mathsf{nil},ws)\to ws)=\{\ \widetilde{\mathsf{rev}}(xs,\mathsf{nil})\twoheadrightarrow\mathsf{tp}_2(xs,\mathsf{nil})\ \}\\&\mathcal{I}nv_{\mathsf{rule}}(\mathsf{rev}(\mathsf{cons}(x,xs),ys)\to\mathsf{rev}(xs,\mathsf{cons}(x,ys)))\\&\qquad=\{\ \widetilde{\mathsf{rev}}(xs,\mathsf{cons}(x,ys))\to\widetilde{\mathsf{rev}}(\mathsf{cons}(x,xs),ys)\ \}\end{aligned}$$

Thus, R_{reverse} is inverted as follows:

$$\mathcal{I}nv(R_{\mathsf{reverse}}) = \left\{ \begin{array}{r} \mathsf{reverse}^{-1}(ys) \to \mathsf{tp}_1(xs) \Leftarrow ys \twoheadrightarrow zs;\ \widetilde{\mathsf{rev}}(\mathsf{nil}, zs) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil}) \\ \widetilde{\mathsf{rev}}(xs, \mathsf{nil}) \to \mathsf{tp}_2(xs, \mathsf{nil}) \\ \widetilde{\mathsf{rev}}(xs, \mathsf{cons}(x, ys)) \to \widetilde{\mathsf{rev}}(\mathsf{cons}(x, xs), ys) \end{array} \right\}$$

$\mathcal{I}nv(R_{\mathsf{reverse}})$ is non-overlapping and operationally terminating.

Now, we show the correctness of $\mathcal{I}nv$.

▶ **Lemma 3.9** (completeness). *Let R be a* tp*-free constructor TRS over a signature $\mathcal{F}$, f be a defined symbol of R, and c be $\mathcal{I}nv_{\mathsf{c}}(f(u_1, \cdots, u_n) \twoheadrightarrow x)$ that is needed for generating $\mathcal{I}nv(R)$, where x is a fresh variable. For any constructor substitution σ, if $f(u_1, \cdots, u_n)\sigma \xrightarrow{*}_{\mathsf{c}, \mathcal{N}(R)} x\sigma$, then there exists an extended constructor substitution σ' of σ such that $s\sigma' \xrightarrow{*}_{\mathsf{c}, \mathcal{I}nv(R)} t\sigma'$ for each condition $s \twoheadrightarrow t$ in c.*

Proof (Sketch). This can be proved by induction on the length n of $\xrightarrow{n}_{\mathsf{c}, \mathcal{N}(R)}$ (see [25]). ◀

▶ **Lemma 3.10** (soundness). *Let R be a* tp*-free constructor TRS over a signature $\mathcal{F}$, f be a defined symbol of R, and c be $\mathcal{I}nv_{\mathsf{c}}(f(u_1, \cdots, u_n) \twoheadrightarrow x)$ that is needed for generating $\mathcal{I}nv(R)$, where x is a fresh variable. For any constructor substitution σ, if $s\sigma \xrightarrow{*}_{\mathsf{c}, \mathcal{I}nv(R)} t\sigma$ for each condition $s \twoheadrightarrow t$ in c, then $f(u_1, \cdots, u_n)\sigma \xrightarrow{*}_{\mathsf{c}, \mathcal{N}(R)} x\sigma$.*

Proof (Sketch). This lemma can be proved by induction on the lexicographic product (m, n) where m and n are the maximum integers among (m', n') of $s\sigma \xrightarrow{n'}_{\mathsf{c}(m'), \mathcal{I}nv(R)} t\sigma$ (see [25]). ◀

▶ **Theorem 3.11** (correctness). *Let R be a* tp*-free constructor TRS over a signature $\mathcal{F}$, and f be a defined symbol of R. Then, all of the following hold:*

- *If f is not tail recursive, then f^{-1} in $\mathcal{I}nv(R)$ is a full inverse of f.*
- *If f is tail recursive, then $\widetilde{f}$ in $\mathcal{I}nv(R)$ is a tail recursive inverse of f.*

Proof (Sketch). This theorem follows from Lemmas 3.6, 3.9 and 3.10. ◀

Now, we show two simple optimizations that can be applied after $\mathcal{I}nv$:

1. A rule $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots, p_1 \twoheadrightarrow p_2; \cdots; s_k \twoheadrightarrow t_k$ can be replaced by $l\sigma \to r\sigma \Leftarrow s_1\sigma \twoheadrightarrow t_1\sigma; \cdots; s_k\sigma \twoheadrightarrow t_k\sigma$ if p_1 and p_2 are unifiable constructor terms, where σ is a most general unifier of p_1 and p_2 such that $\mathcal{VR}an(\sigma) \cap \mathcal{V}ar(\rho) = \emptyset$.
2. The rule ρ above can be removed from the rewrite system if p_1 and p_2 are not unifiable.

For a constructor TRS R, the CTRS obtained by applying the above two optimizations to $\mathcal{I}nv(R)$ as much as possible is denoted by $\mathcal{I}nv_{\mathsf{opt}}(R)$.

▶ **Example 3.12.** $\mathcal{I}nv(R_{\mathsf{reverse}})$ in Example 3.8 is optimized as follows:

$$\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{reverse}}) = \left\{ \begin{array}{r} \mathsf{reverse}^{-1}(ys) \to \mathsf{tp}_1(xs) \Leftarrow \widetilde{\mathsf{rev}}(\mathsf{nil}, ys) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil}) \\ \widetilde{\mathsf{rev}}(xs, \mathsf{nil}) \to \mathsf{tp}_2(xs, \mathsf{nil}) \\ \widetilde{\mathsf{rev}}(xs, \mathsf{cons}(x, ys)) \to \widetilde{\mathsf{rev}}(\mathsf{cons}(x, xs), ys) \end{array} \right\}$$

As mentioned before, the technique for the elimination of non-determinism of [6, 7, 11] is able to produce a similar result starting from the full inverse system obtained by the old approach (i.e., the system shown in Example 3.1). However, their technique is very sensitive to the structure of the program and does not always produce non-overlapping deterministic programs (e.g., the next example cannot be inverted following the approach of [6, 7, 11]).

Our next example illustrates the application of the inversion technique when there are several initial call patterns.

▶ **Example 3.13.** Consider the following TRS $R_{\mathsf{treepaths}}$:

$$R_{\mathsf{treepaths}} = \left\{ \begin{array}{r} \mathsf{treepaths}(t) \to \mathsf{paths}(t, \mathsf{nil}, \mathsf{nil}) \\ \mathsf{paths}(\mathsf{leaf}, qs, qss) \to \mathsf{cons}(qs, qss) \\ \mathsf{paths}(\mathsf{bin}(l, r), ps, pss) \to \mathsf{paths}(l, \mathsf{cons}(0, ps), \mathsf{paths}(r, \mathsf{cons}(1, ps), pss)) \end{array} \right\}$$

The function treepaths takes a binary tree over bin and leaf as input and returns the list of paths from the root to leaves, e.g., for $\mathsf{bin}(\mathsf{leaf}, \mathsf{bin}(\mathsf{bin}(\mathsf{leaf}, \mathsf{leaf}), \mathsf{bin}(\mathsf{bin}(\mathsf{leaf}, \mathsf{leaf}), \mathsf{leaf})))$, the function treepaths returns the list corresponding to `[[0],[0,0,1],[1,0,1],[0,0,1,1],[1,0,1,1],[1,1,1]]`, where 0 and 1 indicate the left and right children, resp. The right-hand side of the third rule contains a nested function call to paths, so it is an initial call of paths. Thus, $R_{\mathsf{treepaths}}$ is inverted as follows:

$$\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{treepaths}}) = \left\{ \begin{array}{r} \mathsf{treepaths}^{-1}(\mathsf{cons}(qs, qss)) \to \mathsf{tp}_1(t) \Leftarrow \widetilde{\mathsf{paths}}(\mathsf{leaf}, qs, qss) \twoheadrightarrow \mathsf{tp}_3(t, \mathsf{nil}, \mathsf{nil}) \\ \widetilde{\mathsf{paths}}(t, \mathsf{nil}, \mathsf{nil}) \to \mathsf{tp}_3(t, \mathsf{nil}, \mathsf{nil}) \\ \widetilde{\mathsf{paths}}(r, \mathsf{cons}(1, ps), pss) \to \mathsf{tp}_3(r, \mathsf{cons}(1, ps), pss) \\ \widetilde{\mathsf{paths}}(l, \mathsf{cons}(0, ps), \mathsf{cons}(qs, qss)) \to \widetilde{\mathsf{paths}}(\mathsf{bin}(l, r), ps, pss) \\ \Leftarrow \widetilde{\mathsf{paths}}(\mathsf{leaf}, qs, qss) \twoheadrightarrow \mathsf{tp}_3(r, \mathsf{cons}(1, ps), pss)) \end{array} \right\}$$

▶ **Example 3.14.** Consider the TRS $R_{\mathsf{unbin2}} = R_{\mathsf{ub}} \cup R_{\mathsf{inc}}$ (a slight variation of that in [6, 7, 11]) with

$$R_{\mathsf{ub}} = \left\{ \begin{array}{r} \mathsf{unbin}(u) \to \mathsf{ub}(u, \mathsf{nil}) \\ \mathsf{ub}(\mathsf{s}(\mathsf{zero}), b) \to b \\ \mathsf{ub}(\mathsf{s}(\mathsf{s}(v)), b) \to \mathsf{ub}(\mathsf{s}(v), \mathsf{inc}(b)) \end{array} \right\}; \; R_{\mathsf{inc}} = \left\{ \begin{array}{r} \mathsf{inc}(\mathsf{nil}) \to \mathsf{cons}(0, \mathsf{nil}) \\ \mathsf{inc}(\mathsf{cons}(0, xs)) \to \mathsf{cons}(1, xs) \\ \mathsf{inc}(\mathsf{cons}(1, xs)) \to \mathsf{cons}(0, \mathsf{inc}(xs)) \end{array} \right\}$$

The function unbin translates natural numbers $\mathsf{s}(\mathsf{zero})$, $\mathsf{s}(\mathsf{s}(\mathsf{zero}))$, $\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{zero})))$, $\cdots$ into the (reversed) binary numbers nil, $\mathsf{cons}(0, \mathsf{nil})$, $\mathsf{cons}(1, \mathsf{nil})$, $\mathsf{cons}(0, \mathsf{cons}(0, \mathsf{nil}))$, $\cdots$, resp., where nil represents 1 and inc is used to increment binary numbers. R_{unbin2} is inverted as follows:

$$\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}}) = \left\{ \begin{array}{r} \mathsf{unbin}^{-1}(b) \to \mathsf{tp}_1(u) \Leftarrow \widetilde{\mathsf{ub}}(\mathsf{s}(\mathsf{zero}), b) \twoheadrightarrow \mathsf{tp}_2(u, \mathsf{nil}) \\ \widetilde{\mathsf{ub}}(u, \mathsf{nil}) \to \mathsf{tp}_2(u, \mathsf{nil}) \\ \widetilde{\mathsf{ub}}(\mathsf{s}(v), x) \to \widetilde{\mathsf{ub}}(\mathsf{s}(\mathsf{s}(v)), b) \Leftarrow \mathsf{inc}^{-1}(x) \twoheadrightarrow \mathsf{tp}_1(b) \end{array} \right\} \cup \mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{inc}})$$

where

$$\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{inc}}) = \left\{ \begin{array}{r} \mathsf{inc}^{-1}(\mathsf{cons}(0, \mathsf{nil})) \to \mathsf{tp}_1(\mathsf{nil}) \\ \mathsf{inc}^{-1}(\mathsf{cons}(1, xs)) \to \mathsf{tp}_1(\mathsf{cons}(0, xs)) \\ \mathsf{inc}^{-1}(\mathsf{cons}(0, ys)) \to \mathsf{tp}_1(\mathsf{cons}(1, xs)) \Leftarrow \mathsf{inc}^{-1}(ys) \twoheadrightarrow \mathsf{tp}_1(xs) \end{array} \right\}$$

The result is operationally terminating and overlapping while the method in [6, 7, 11] fails to generate an inverse of unbin due to a so-called *shift/shift* conflict. Moreover, the application of the non-determinism elimination in [6, 7, 11] to $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}})$ also fails due to a shift/shift conflict.

Unfortunately, although $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}})$ is operationally terminating, its rules are overlapping. Luckily, there exists a simple approach to improve the result of the inversion transformation.

If a CTRS is non-overlapping, then the application of rewrite rules to innermost redexes is deterministic. For analyzing confluence of CTRSs, the notion of *infeasible* critical pairs is

useful since overlapping rules producing infeasible critical pairs can be considered to be non-overlapping in practice [33]. However, this notion is not so helpful to determine which rules should be applied to redexes before evaluating the conditional parts. On the other hand, a bit more attentive analysis is sometimes helpful. Consider $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}})$ in Example 3.14 again. The critical pairs of $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}})$ are the following:

$$\begin{array}{llll}
(& \mathsf{tp}_2(\mathsf{s}(v), \mathsf{nil}), & \widetilde{\mathsf{ub}}(\mathsf{s}(\mathsf{s}(v)), b) &) \Leftarrow \mathsf{inc}^{-1}(\mathsf{nil}) \twoheadrightarrow \mathsf{tp}_1(b) \\
(& \widetilde{\mathsf{ub}}(\mathsf{s}(\mathsf{s}(v)), b), & \mathsf{tp}_2(\mathsf{s}(v), \mathsf{nil}) &) \Leftarrow \mathsf{inc}^{-1}(\mathsf{nil}) \twoheadrightarrow \mathsf{tp}_1(b) \\
(& \mathsf{tp}_1(\mathsf{nil}), & \mathsf{tp}_1(\mathsf{cons}(1, xs)) &) \Leftarrow \mathsf{inc}^{-1}(\mathsf{nil}) \twoheadrightarrow \mathsf{tp}_1(xs) \\
(& \mathsf{tp}_1(\mathsf{cons}(1, xs)), & \mathsf{tp}_1(\mathsf{nil}) &) \Leftarrow \mathsf{inc}^{-1}(\mathsf{nil}) \twoheadrightarrow \mathsf{tp}_1(xs)
\end{array}$$

Since $\mathsf{inc}^{-1}(\mathsf{nil})$ is undefined, it is easy to see that these critical pairs are infeasible and that we do not need to apply the second rule of $\widetilde{\mathsf{ub}}$ and the third rule of inc^{-1} to terms of the forms $\widetilde{\mathsf{ub}}(u, \mathsf{nil})$ and $\mathsf{inc}^{-1}(\mathsf{cons}(0, \mathsf{nil}))$, resp. This observation comes from the performance of the restricted completion in [21] that is used for transforming *unraveled* inverted TRSs into confluent ones. We call a critical pair $(u, v) \Leftarrow c$ *statically infeasible w.r.t.* $\underset{\mathsf{c}}{\rightarrow}$ if c contains a condition $f(s_1, \cdots, s_n) \twoheadrightarrow t$ such that $s_1, \cdots, s_n, t$ are constructor terms and $f(s_1, \cdots, s_n)$ is unifiable with none of the left-hand sides of f-rules. We call a CTRS *practically non-overlapping* if every critical pair of the CTRS is either trivial and at least one of the rules forming the critical pair is unconditional, or it is statically infeasible. For a practically non-overlapping constructor CTRS, we can straightforwardly give a priority between its rewrite rules in advance as *priority rewrite systems* [29] that are more faithful to functional programs.

▶ **Example 3.15.** Consider $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}})$ in Example 3.14 again. All the critical pairs are statically infeasible and thus $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}})$ is practically non-overlapping.

When removing the assumption that a tail recursive function has a single non-recursive rule, we extend Definition 3.7 as follows: the condition sequence $\mathcal{I}nv_{\mathsf{c}}(s_k \twoheadrightarrow t_k); \cdots; \mathcal{I}nv_{\mathsf{c}}(s_1 \twoheadrightarrow t_1)$ is replaced by $c_1; \cdots; c_k$ where $c_i \in \mathcal{I}nv_{\mathsf{c}}(s_i \twoheadrightarrow t_i)$ for all $1 \leq i \leq k$, and Condition inverse (ii) is replaced by $\mathcal{I}nv_{\mathsf{c}}(f(u_1, \cdots, u_n) \twoheadrightarrow x) = \{x \twoheadrightarrow r; \widetilde{f}(v_1, \cdots, v_n) \twoheadrightarrow \mathsf{tp}_n(u_1, \cdots, u_n) \mid f(v_1, \cdots, v_n) \to r$ is a renamed variant of a non-recursive f-rule$\}$. Our next example illustrates the application of this extended inversion technique to tail recursive functions having two non-recursive rules.

▶ **Example 3.16** (reverse2)**.** Consider the following variant of reverse in Example 3.1:

$$R_{\mathsf{reverse2}} = \left\{ \begin{array}{r}
\mathsf{reverse}(xs) \to \mathsf{rev}(xs, \mathsf{nil}) \\
\mathsf{rev}(\mathsf{nil}, ys) \to ys \\
\mathsf{rev}(\mathsf{cons}(x, \mathsf{nil}), ys) \to \mathsf{snoc}(ys, x) \\
\mathsf{rev}(\mathsf{cons}(x_1, \mathsf{cons}(x_2, xs)), ys) \to \mathsf{rev}(xs, \mathsf{cons}(x_2, \mathsf{cons}(x_1, ys))) \\
\mathsf{snoc}(\mathsf{nil}, y) \to \mathsf{cons}(y, \mathsf{nil}) \\
\mathsf{snoc}(\mathsf{cons}(x, xs), y) \to \mathsf{cons}(x, \mathsf{snoc}(xs, y))
\end{array} \right\}$$

Applying the extended inversion, R_{reverse2} is inverted as follows:

$$\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{reverse2}}) = \left\{ \begin{array}{r}
\mathsf{reverse}^{-1}(ys) \to \mathsf{tp}_1(xs) \Leftarrow \widetilde{\mathsf{rev}}(\mathsf{nil}, ys) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil}) \\
\mathsf{reverse}^{-1}(ys) \to \mathsf{tp}_1(xs) \Leftarrow \mathsf{snoc}^{-1}(ys) \twoheadrightarrow \mathsf{tp}_2(zs, z); \\
\qquad \widetilde{\mathsf{rev}}(\mathsf{cons}(z, \mathsf{nil}), zs) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil}) \\
\widetilde{\mathsf{rev}}(xs, \mathsf{nil}) \to \mathsf{tp}_2(xs, \mathsf{nil}) \\
\widetilde{\mathsf{rev}}(xs, \mathsf{cons}(x_2, \mathsf{cons}(x_1, ys))) \to \widetilde{\mathsf{rev}}(\mathsf{cons}(x_1, \mathsf{cons}(x_2, xs)), ys) \\
\mathsf{snoc}^{-1}(\mathsf{cons}(y, \mathsf{nil})) \to \mathsf{tp}_2(\mathsf{nil}, y) \\
\mathsf{snoc}^{-1}(\mathsf{cons}(x, ys)) \to \mathsf{tp}_2(\mathsf{cons}(x, xs), y) \Leftarrow \mathsf{snoc}^{-1}(ys) \twoheadrightarrow \mathsf{tp}_2(xs, y)
\end{array} \right\}$$

The result is operationally terminating but not practically non-overlapping while the method in [6, 7, 11] fails to generate an inverse of reverse2 due to a shift/shift conflict. Unfortunately, the application of the non-determinism elimination in [6, 7, 11] to $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{reverse2}})$ also fails due to a shift/shift conflict.

4 Comparison with Previous Approaches

The method in this paper is a conservative extension of [20, 23] to better deal with the inversion of tail recursive functions. While the previous approach always produces non-terminating rules from the inversion of tail recursive functions, this is not always the case for the new approach, which is thus strictly better regarding the generation of terminating systems. Note that the method in this paper can invert every constructor TRS, though the resulting CTRS is not always practically non-overlapping and operationally terminating.

In the following, we show some experimental results from the evaluation of our new approach. First, we applied it to the standard 15 benchmarks from [10].[2] Ten of the benchmarks are non-tail-recursive functions without tail-recursive rules. For these benchmarks, our method generates the same inverse systems as the previous method [20, 24, 23] and all the results are operationally terminating and practically non-overlapping. Note that the method in [6, 7, 11] is also successful for the 10 benchmarks.

As for the remaining 5 benchmarks, three of them are tail recursive ones, and the other two are non-tail recursive functions containing tail-recursive rules. Table 1 summarizes the results on the 3 tail recursive functions, reverse, unbin and treepaths, and also contains the results on other tail recursive functions: unbin2 (Example 3.14) and reverse2 (Example 3.16). Moreover, the lower half of Table 1 summarizes the results of applying the inversion to inverses obtained from the first 3 benchmarks: $\mathcal{I}nv_{\mathsf{opt}}(\mathsf{reverse})$, $\mathcal{I}nv_{\mathsf{opt}}(\mathsf{unbin})$ and $\mathcal{I}nv_{\mathsf{opt}}(\mathsf{treepaths})$ are the systems obtained by applying $\mathcal{I}nv_{\mathsf{opt}}$ to reverse, unbin and treepaths, resp., and lrinv-reverse, lrinv-unbin and lrinv-treepaths are obtained by applying the method [6, 7, 11] to reverse, unbin and treepaths, respectively. Note that the benchmarks in the second half are in principle out of scope of our approach, but we applied our inversion method to them extending it straightforwardly. Operational termination of the resulting systems obtained by our method was proved by the termination tool VMTL [30].

By lack of space, we did not show a transformation of non-tail-recursive functions to equivalent tail-recursive ones, that is an extension of the idea of *continuation passing style* to the first-order setting. However, after transforming the remaining two benchmarks that are not tail-recursive but contain tail-recursive rules into tail-recursive ones, our method succeeds in generating operationally terminating and practically non-overlapping inverse systems. Thus, our method is successful for all the benchmarks shown in [6, 7, 11]. A similar transformation is called *context-moving* [5]. However, this transformation is sound when the contexts to be moved satisfy a property like associativity or commutativity. Thus, the context-moving transformation is not applicable to arbitrary non-tail-recursive functions.

There exist some examples that the method in [6, 7, 11] fails to invert but our approach succeeds, e.g., unbin2. On the other hand, we have never found an example in which the non-determinism problem cannot be solved by our approach but can be solved by the method in [6, 7, 11], though it may exist. To summarize, a promising strategy for program inversion

[2] Unfortunately, the site shown in [10] is not accessible now. Some of the benchmarks can be found in [6, 7, 11]. All the benchmarks are reviewed in [21] and also available from the following URL for the implementation of our method: `http://www.trs.cm.is.nagoya-u.ac.jp/repius/`.

Table 1 Comparison with the previous approaches [20, 24] and [6, 7, 11]: "SN", "NOV" and "PNOV" mean that the inverse is "operationally terminating", "non-overlapping" and "practically non-overlapping", respectively.

	inverse by [20, 24, 23]	inverse by [6, 7, 11]	inverse by this paper
benchmark	SN?/NOV?/PNOV?	SN?/NOV?	SN?/NOV?/PNOV?
reverse [10, 6, 7, 11]	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
unbin [10, 6, 7, 11]	no/no/no	**yes**/**yes**	**yes**/no/**yes**
treepaths [10, 6, 7, 11]	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
unbin2 (Example 3.14)	no/no/no	fail (no output)	**yes**/no/**yes**
reverse2 (Example 3.16)	no/no/no	fail (no output)	**yes**/no/no
$\mathcal{I}nv_{\mathsf{opt}}$(reverse)	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
$\mathcal{I}nv_{\mathsf{opt}}$(unbin)	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
$\mathcal{I}nv_{\mathsf{opt}}$(treepaths)	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
lrinv-reverse	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
lrinv-unbin	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
lrinv-treepaths	no/no/no	fail (no output)	**yes**/**yes**/**yes**

would first apply our approach and, then, the non-determinism elimination of [6, 7, 11] (appropriately extended to deal with erasing rules), when the output of our approach contains some non-determinism.

As stated before, the idea of our approach to inversion of tail recursive functions is similar to that in [18], though there the idea is only illustrated by using some examples and there is no formal transformation nor correctness proof. On the other hand, we introduced a formal definition and its correctness proof, together with an analysis on overlaps between generated rewrite rules. Moreover, our formalization can be easily extended to tail recursive functions with several initial calls and non-recursive rules.

Another related work is the method in [15], though it is more related to inverse computation than to program inversion. Nevertheless, we note that this method is only applicable to linear functions and it is not guaranteed the termination of the produced programs for any input (only for the inputs that are original outputs). Moreover, tail recursive functions are out of scope of this paper.

5 Conclusion

In this paper, we have proposed the notion of tail recursive inverses and have introduced an appropriate approach to program inversion which extends the previous technique from [20, 23] in order to deal with systems containing tail recursive functions. As mentioned before, for the sake of readability, we assumed the following restrictions in this paper: the original TRS is unconditional, non-erasing and tp-free; and tail recursive functions are self-recursive. However, it would not be difficult to remove these assumptions (and, indeed, they are not required in the implemented method). Moreover, it would be easy to combine the new approach to the inversion of tail recursive functions with the method of [20, 24] for partial inversion, so there is ample room for improving and extending our approach.

As for future work, we plan to develop a method for the elimination of non-determinism in inverted rules since there are examples, such as reverse2 in Example 3.16. Another promising direction for future work is the search of sufficient conditions for tail recursive functions so that the computed inverses with our technique are operationally terminating.

Acknowledgements

We thank the anonymous reviewers for their useful comments to improve this paper. This work has been partially supported by MEXT KAKENHI #21700011 and the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02. Part of this research was done while the first author was visiting the MiST group at the *Universitat Politècnica de València* as part of an Institutional Program for Young Researcher Overseas Visits. The first author is grateful to the members of the MiST and ELP groups.

References

1 S. M. Abramov and R. Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193–229, 2002.

2 J. M. Almendros-Jiménez and G. Vidal. Automatic partial inversion of inductively sequential functions. In *Proc. of the 18th Int'l Symposium on Implementation and Application of Functional Languages (Revised Selected Papers)*, volume 4449 of *Lecture Notes in Computer Science*, pp. 253–270, Springer, 2007.

3 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, United Kingdom, 1998.

4 N. Dershowitz and S. Mitra. Jeopardy. In *Proc. of the 10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, pp. 16–29, Trento, Italy, July 1999.

5 J. Giesl. Context-moving transformations for function verification. In *Selected Papers of the 9th International Workshop on Logic Programming Synthesis and Transformation*, volume 1817 of *Lecture Notes in Computer Science*, pp. 293–312, Springer, 2000.

6 R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In *Proc. of the 1st Asian Symposium on Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*, pp. 246–264, Springer, 2003.

7 R. Glück and M. Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundamenta Informaticae*, 66(4):367–395, 2005.

8 B. Gramlich. On the (non-)existence of least fixed points in conditional equational logic and conditional rewriting. In *Proc. of the 2nd International Workshop on Fixed Points in Computer Science – Extended Abstracts*, pp. 38–40, Paris, France, July 2000.

9 P. G. Harrison. Function inversion. In *Proc. of IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pp. 153–166, North-Holland, 1988.

10 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.

11 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*, volume 3350 of *Lecture Notes in Computer Science*, pp. 219–234, Springer, Jan. 2005.

12 H. Khoshnevisan and K. M. Sephton. InvX: An automatic function inverter. In *Proc. of the 3rd International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pp. 564–568, Springer, 1989.

13 S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95(4):446–453, 2005.

14 M. Marchiori. Unravelings and ultra-properties. In *Proc. of the 5th International Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pp. 107–121, Springer, 1996.

15 K. Matsuda, S.-C. Mu, Z. Hu, and M. Takeichi. A grammar-based approach to invertible programs. In *Proc. of the 19 the European Symposium on Programming*, volume 6012 of *Lecture Notes in Computer Science*, pp. 448–467, Springer, 2010.

16 J. McCarthy. The inversion of functions defined by Turing machines. In *Automata Studies*, pp. 177–181. Princeton University Press, 1956.

17 T. Æ. Mogensen. Semi-inversion of guarded equations. In *Proc. of the 4th International Conference on Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pp. 189–204, Springer, 2005.

18 T. Æ. Mogensen. Report on an implementation of a semi-inverter. In *Proc. of the 6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, volume 4378 of *Lecture Notes in Computer Science*, pp. 322–334, Springer, 2006.

19 T. Æ. Mogensen. Semi-inversion of functional parameters. In *Proc. of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 21–29, New York, NY, USA, 2008. ACM Press.

20 N. Nishida. *Transformational Approach to Inverse Computation in Term Rewriting*. Doctor thesis, Graduate School of Engineering, Nagoya University, Nagoya, Japan, Jan. 2004.

21 N. Nishida and M. Sakai. Completion after program inversion of injective functions. In *Proc. of the 8th International Workshop on Reduction Strategies in Rewriting and Programming*, Volume 237 of *Electronic Notes in Theoretical Computer Science*, pp. 39–56, Apr. 2009.

22 N. Nishida, M. Sakai, and T. Sakabe. Generation of inverse term rewriting systems for pure treeless functions. In *Proc. of the International Workshop on Rewriting in Proof and Computation*, pp. 188–198, Sendai, Japan, Oct. 2001.

23 N. Nishida, M. Sakai, and T. Sakabe. Generation of inverse computation programs of constructor term rewriting systems. *IEICE Transactions on Information and Systems*, J88-D-I(8):1171–1183, Aug. 2005 (in Japanese).

24 N. Nishida, M. Sakai, and T. Sakabe. Partial inversion of constructor term rewriting systems. In *Proc. of the 16th Int'l Conference on Rewriting Techniques and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pp. 264–278, Springer, Apr. 2005.

25 N. Nishida and G. Vidal. Program inversion for tail recursive functions. The full version of this paper, available from `http://www.trs.cm.is.nagoya-u.ac.jp/~nishida/rta11/`.

26 E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, April 2002.

27 A. Romanenko. The generation of inverse functions in Refal. In *Proc. of IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pp. 427–444, North-Holland, 1988.

28 A. Romanenko. Inversion and metacomputation. In *Proc. of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26 of *SIGPLAN Notices*, pp. 12–22, ACM Press, Sept. 1991.

29 M. Sakai and Y. Toyama. Semantics and strong sequentiality of priority term rewriting systems. *Theoretical Computer Science*, 208(1-2):87–110, 1998.

30 F. Schernhammer and B. Gramlich. VMTL—a modular termination laboratory. In *Proc. of the 20th International Conference on Rewriting Techniques and Applications*, volume 5595 of *Lecture Notes in Computer Science*, pp. 285–294, Springer, 2009.

31 P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic*, 11(1):52 pages, Oct. 2009.

32 J. P. Secher and M. H. Sørensen. From checking to inference via driving and dag grammars. In *Proc. of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, volume 37 of *SIGPLAN Notices*, pp. 41–51, Portland, Oregon, USA, March 2002.

33 Y. Takahashi, M. Sakai, and Y. Toyama. On the confluence property of conditional term rewriting systems. *IEICE Transactions*, J79-D-I(11):897–902, 1996 (in Japanese).