Semi-inversion of Conditional Constructor Term Rewriting Systems

Maja Hanne Kirkeby[1(✉)] and Robert Glück[2]

[1] Roskilde University, Roskilde, Denmark
kirkebym@acm.org
[2] DIKU, University of Copenhagen, Copenhagen, Denmark
glueck@acm.org

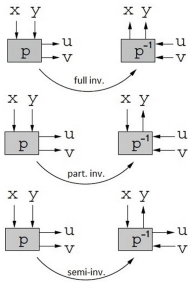
Abstract. Inversion is an important and useful program transformation and has been studied in various programming language paradigms. Semi-inversion is more general than just swapping the input and output of a program; instead, parts of the input and output can be freely swapped. In this paper, we present a polyvariant semi-inversion algorithm for conditional constructor term rewriting systems. These systems can model logic and functional languages, which have the advantage that semi-inversion, as well as partial and full inversion, can be studied across different programming paradigms. The semi-inverter makes use of local inversion and a simple but effective heuristic and is proven to be correct. A Prolog implementation is applied to several problems, including inversion of a simple encrypter and of a program inverter for a reversible language.

Keywords: Program transformation · Program inversion · Conditional term rewriting systems · Logic and functional programs

1 Introduction

Programs that are inverse to each other are widely used, such as encoding and decoding of data. The transformation of an encoder into a decoder, or vice versa, is called full inversion. *Semi-inversion*, the most general type of program inversion, transforms one relation into a new relation that takes a subset of the original input and output as the new input. For example, the transformation of a symmetric encrypter into a decrypter cannot be achieved by conventional full inversion because both programs take the same key as input.

In this paper, we present a polyvariant semi-inversion algorithm for an oriented *conditional constructor term rewriting system* (CCS) [22]. The algorithm makes use of local inversion and a simple but effective heuristic and is proven to be correct. A Prolog implementation is applied to several transformation problems, including the inversion of a simple symmetric encrypter. As a special transformation challenge, a program inverter for a reversible imperative language was inverted into a copy of itself modulo variable renaming.



M. Gabbrielli (Ed.): LOPSTR 2019, LNCS 12042, pp. 243–259, 2020.
https://doi.org/10.1007/978-3-030-45260-5_15

We distinguish between three forms of program inversion: *Full inversion* turns a program $\mathtt{p}$ into a new program $\mathtt{p}^{-1}$, where the original inputs and outputs are exchanged. If $\mathtt{p}$ is injective, then $\mathtt{p}^{-1}$ implements a function. *Partial inversion* yields a program $\mathtt{p}^{-1}$ that inputs the original output ($\mathtt{u}$, $\mathtt{v}$) and some of the original input ($\mathtt{x}$) and then returns the remaining input ($\mathtt{y}$). *Semi-inversion* yields a program $\mathtt{p}^{-1}$ that, given some of the original input ($\mathtt{x}$) and some of the original output ($\mathtt{v}$), returns the remaining input ($\mathtt{y}$) and output ($\mathtt{u}$). The programs $\mathtt{p}$ and $\mathtt{p}^{-1}$ may implement functions or more general relations. Full inversion is a subproblem of partial inversion, which is a subproblem of semi-inversion:

$$\text{full inversion} \subseteq \text{partial inversion} \subseteq \text{semi-inversion}.$$

Dijkstra was the first to study the full inversion of programs in a guarded command language [6]. Subsequently, some program inversion algorithms were developed for different forms of inversion and for different programming languages. Of those, only Nishida et al. [19] and Almendros-Jiménez et al. [2] have considered term rewriting systems, where the latter constrained the systems such that terms have a unique normal form, i.e., the systems express functional input-output relations. Mogensen [14,15], who developed the first semi-inversion algorithm, did so for a deterministic guarded equational language, i.e., with functional input-output relations. Methods for inversion have been studied in the context of functional languages [7–11]. The motivation for using *program inversion* instead of *inverse interpretation*, such as [1,13], is similar to the motivation for using translation instead of interpretation.

The main advantage of oriented conditional constructor term rewriting systems is that they can model both *logic* and *functional* languages [5], and, hence, *functional logical* languages [12]. When modeling logic languages, an efficient evaluation requires narrowing (unification) [5,12], which typically has a larger search space than standard rewriting (matching). By requiring that the rewrite rules be not only left-orthogonal but also right-orthogonal and non-deleting, we can also model *reversible* languages [23]. This enables us to focus on the essence of semi-inversion without considering language-specific details. The semantics of *functions* and *relations* can be expressed and efficiently calculated in the same formalism (*cf.*, Ex. 2). The idea to use CCSs to investigate semi-inversion for different language paradigms was inspired by the partial inverter developed by Nishida et al. [19].

The new semi-inverter relates to some of the mentioned inversion algorithms:

	Functions	Relations
Full inversion	Glück and Kawabe [8]	Nishida et al. [17,20]
Partial inversion	Almendros-Jiménez et al. [2]	Nishida et al. [19]
Semi-inversion	Mogensen [14]	*This algorithm*

This paper provides (1) a polyvariant semi-inversion algorithm for CCSs that uses local inversion and is proven correct; (2) a simple but effective heuristic to avoid narrowing and to minimize the search space; and (3) an experimental evaluation by applying the Prolog implementation to a simple encryption

algorithm, a physical discrete-event simulation, and a program inverter for a reversible language.

Overview: After an brief overview of the semi-inverter (Sect. 2), we formally define conditional term rewriting systems and semi-inversion (Sect. 3). Then, we present our algorithm (Sect. 4) and report on the experimental results (Sect. 5).[1]

2 The Semi-inversion Algorithm—An Overview

This section gives an informal overview of semi-inversion and illustrates the semi-inversion algorithm with a short, familiar example. Both semi-inversion and the algorithm will be formalized and defined in the following sections.

We let semi-inversion cover *rewritings* of the form

$$f(s_1, \ldots, s_n) \;\to^*_{\mathcal{R}}\; \langle t_1, \ldots, t_m \rangle,$$

where f is an n-ary *function symbol* with co-arity m defined in the conditional term rewriting system $\mathcal{R}$; *input and output terms* $s_1, \ldots, s_n$ and $t_1, \ldots, t_m$ are ground constructor terms; and $\langle ... \rangle$ is a special m-ary constructor containing the m output terms. The transformation of f into a semi-inverse $\underline{f}$ is w.r.t. indices of known input and output terms. If we assume that the first a input and b output terms are the known arguments, then the semi-inverse $\underline{f}_{\{1,\ldots,a\}\{1,\ldots,b\}}$ takes the form

$$\underline{f}_{\{1,\ldots,a\}\{1,\ldots,b\}}(s_1, \ldots, s_a, t_1, \ldots, t_b) \;\to^*_{\underline{\mathcal{R}}}\; \langle s_{a+1}, \ldots, s_n, t_{b+1}, \ldots, t_m \rangle.$$

The *input-output index sets* $\{1, \ldots, a\}$ and $\{1, \ldots, b\}$ label the new function symbol $\underline{f}$ and serve to distinguish different semi-inverses of the same f. The semi-inversion algorithm locally inverts each rule needed for the rewriting sequence in the *semi-inverted rewriting system* $\underline{\mathcal{R}}$ such that the known parameters specified by the two index sets occur on the left-hand side and all others occur on the right-hand side of the semi-inverted rules of f. Semi-inversion is *polyvariant* because $\underline{\mathcal{R}}$ may include several different semi-inversions of the rules defining f in $\mathcal{R}$, while, in contrast, full inversion is *monovariant*, as it requires only one variant per f.

Example 1. Take as an example the multiplication $x \cdot y = z$ of two unary numbers x and y defined by adding y x times (`s1`–`s4`, Fig. 1), which is similar to the unconditional system (`r1`–`r4`, Fig. 1) suggested by [19]. Inversion of multiplication `mul` is w.r.t. the second input y and the first (and only) output z. That is, input-output index sets $I = \{2\}$ and $O = \{1\}$ yield the rewrite rules for division $z/y = x$ and, as a subtask, partially inverts addition $x + y = z$ into subtraction $z - x = y$ (`t1`–`t4`, as shown in Fig. 1). Multiplication `mul(x,y)` and addition `add(x,y)` are inverted into division $\underline{\texttt{mul}}_{\{2\}\{1\}}$`(y,z)` and subtraction $\underline{\texttt{add}}_{\{1\}\{1\}}$`(x,z)` and are replaced by the forms `div(z,y)` and `sub(z,x)` for readability. Some of

[1] The extended abstract of a talk, Nordic Workshop on Programming Theory, Univ. of Bergen, Dept. of Informatics, Report 403, 2012, is partially used in Sects. 1 and 5.

the inverted rules have a conditional part (the conjunction to the right of $\Leftarrow$), which must be satisfied to apply a rule and may bind variables, *e.g.*, `y`, in rule `t4`.

The algorithm is illustrated in Fig. 2 by the stepwise inversion of rule `s2`. First, all function symbols are labeled with index sets, starting with the given index sets on the left-hand side and then repeatedly (from left to right) labeling the function symbols in the conditions with indices of the known arguments (Step 1). Variable `w` is known after `add` is rewritten, so the rightmost `mul` is labeled $\texttt{mul}_{\{2\}\{1\}}$. Finally, *local inversion* brings all known parts to the left-hand side according to the index sets (Step 2), e.g., $\texttt{mul}_{\{2\}\{1\}}\texttt{(s(x),y)} \to \langle \texttt{z} \rangle$ into $\underline{\texttt{mul}}_{\{2\}\{1\}}\texttt{(y,z)} \to \langle \texttt{s(x)} \rangle$.

We note that in Fig. 1, division by zero, `div(z,0)`, is undefined (due to infinite rewriting by repeatedly subtracting `0` from `z`). Division of zero, `div(0,y)`, where $\texttt{y} > 0$, correctly defines zero as a result. Both rules `t1` and `t2` match, but only rule `t1` can be applied. The pre-processing and post-processing that transform between unconditional and flat conditional constructor systems are standard techniques and not discussed here; see, *e.g.*, [19,21,22].

Unconditional rules:

$\texttt{r1: mul(0,y)} \to \texttt{0}$ $\quad$ $\texttt{r2: mul(s(x),y)} \to \texttt{add(y,mul(x,y))}$

$\texttt{r3: add(0,y)} \to \texttt{y}$ $\quad$ $\texttt{r4: add(s(x),y)} \to \texttt{s(add(x,y))}$

Flat rules:

$\texttt{s1: mul(0,y)} \to \langle \texttt{0} \rangle$ $\quad$ $\texttt{s2: mul(s(x),y)} \to \langle \texttt{z} \rangle \Leftarrow \texttt{add(y,w)} \to \langle \texttt{z} \rangle \wedge \texttt{mul(x,y)} \to \langle \texttt{w} \rangle$

$\texttt{s3: add(0,y)} \to \langle \texttt{y} \rangle$ $\quad$ $\texttt{s4: add(s(x),y)} \to \langle \texttt{s(z)} \rangle \Leftarrow \texttt{add(x,y)} \to \langle \texttt{z} \rangle$

Inverted rules (after renaming):

$\texttt{t1: div(0,y)} \to \langle \texttt{0} \rangle$ $\quad$ $\texttt{t2: div(z,y)} \to \langle \texttt{s(x)} \rangle \Leftarrow \texttt{sub(z,y)} \to \langle \texttt{w} \rangle \wedge \texttt{div(w,y)} \to \langle \texttt{x} \rangle$

$\texttt{t3: sub(y,0)} \to \langle \texttt{y} \rangle$ $\quad$ $\texttt{t4: sub(s(z),s(x))} \to \langle \texttt{y} \rangle \Leftarrow \texttt{sub(z,x)} \to \langle \texttt{y} \rangle$

Fig. 1. Partial inversion of multiplication into division.

Label all function symbols (index set propagation):

Step 1
$$\begin{cases} \texttt{mul(s(x),y)} \to \langle \texttt{z} \rangle \Leftarrow \texttt{add(y,w)} \to \langle \texttt{z} \rangle \wedge \texttt{mul(x,y)} \to \langle \texttt{w} \rangle \\ \texttt{mul}_{\{2\}\{1\}}\texttt{(s(x),y)} \to \langle \texttt{z} \rangle \Leftarrow \texttt{add}_{\{1\}\{1\}}\texttt{(y,w)} \to \langle \texttt{z} \rangle \wedge \texttt{mul}_{\{2\}\{1\}}\texttt{(x,y)} \to \langle \texttt{w} \rangle \end{cases}$$

Local inversion:

Step 2
$$\begin{cases} \texttt{mul}_{\{2\}\{1\}}\texttt{(s(x),y)} \to \langle \texttt{z} \rangle \Leftarrow \texttt{add}_{\{1\}\{1\}}\texttt{(y,w)} \to \langle \texttt{z} \rangle \wedge \texttt{mul}_{\{2\}\{1\}}\texttt{(x,y)} \to \langle \texttt{w} \rangle \\ \underline{\texttt{mul}}_{\{2\}\{1\}}\texttt{(y,z)} \to \langle \texttt{s(x)} \rangle \Leftarrow \underline{\texttt{add}}_{\{1\}\{1\}}\texttt{(y,z)} \to \langle \texttt{w} \rangle \wedge \underline{\texttt{mul}}_{\{2\}\{1\}}\texttt{(y,w)} \to \langle \texttt{x} \rangle \end{cases}$$

Fig. 2. Stepwise inversion of rule `s2` in Fig. 1 w.r.t. the input-output index sets {2} {1}.

3 Conditional Constructor Systems and Semi-inversion

First, we recall the basic concepts of conditional term rewriting systems following the terminology of Ohlebusch [22] and their ground constructor-based

relation [20]. Then, we define what we call conditional constructor term rewriting systems (CCSs) and describe how they model a series of language paradigms. We also describe the properties for when they can be evaluated efficiently, which relates to the design goals for our semi-inverter. Finally, we define semi-inversion of such systems and give the first insights into the nature of semi-inversion.

3.1 Preliminaries for Conditional Term Rewriting

We assume a countable set of variables $\mathcal{V}$. A finite signature $\mathcal{F}$ is assumed to be partitioned into two disjoint sets: a set of *defined function* symbols $\mathcal{D}$, each $f \in \mathcal{D}$ with an arity n and a co-arity m, written $f/n/m$, and a set of constructor symbols $\mathcal{C}$, each $a \in \mathcal{C}$ with an arity n. We denote the set of all terms over $\mathcal{F}$ and $\mathcal{V}$ by $T(\mathcal{F}, \mathcal{V})$. A term s is a *ground* term if it has no variables, a *constructor* term if it contains no function symbols, and a *ground constructor* term if it is both a ground term and a constructor term. Every subterm s of a term t has at least a position p, and we denote this subterm by $t|_p = s$, with the root symbol denoted $root(t)$. Furthermore, we let $t[s']_p$ denote a new term where the subterm at position p in t is replaced by a new (sub)term s'. A *substitution* σ is a mapping from variables to terms, a *ground substitution* is a mapping from variables to ground terms, and a *constructor substitution* is a mapping from variables to constructor terms.

A *conditional rewrite rule* is of the form $l \to r \Leftarrow c$, where the left-hand side l is a non-variable and $root(l) \in \mathcal{D}$, the right-hand side r is a term, and the *conditions* c are a (perhaps empty) conjunction of conditions $l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k$.

A *conditional term rewriting system* $\mathcal{R}$ *over a signature* $\mathcal{F}$, abbreviated *CTRS*, is a finite set of conditional rewrite rules $l_0 \to r_0 \Leftarrow l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k$ over all terms in $T(\mathcal{F}, \mathcal{V})$ such that the defined functions $\mathcal{D} = \{root(l) \mid l \to r \Leftarrow c \in \mathcal{R}\}$ and constructors $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. The conditions are interpreted as reachability, defining a so-called oriented CTRS, e.g., [22].

A *ground constructor-based rewrite relation* $\to_{\mathcal{R}}$ associated with a CTRS over $\mathcal{F}$ $\to_{\mathcal{R}}$ is the smallest binary relation for a pair of ground terms $s, t \in T(\mathcal{F}, \emptyset)$, where there is a position p, a ground constructor substitution σ and a rewrite rule $l \to r \Leftarrow c$ such that $s|_p = l\sigma$, $s[r\sigma]_p = t$ and, for each condition $(l_i \to r_i) \in c$, $l_i\sigma \to^*_{\mathcal{R}} r_i\sigma$.

3.2 Conditional Constructor Systems

In this study, we focus on a subclass of CTRSs we call *conditional constructor term rewriting systems*. These systems are both input to and output from the semi-inversion algorithm. They are also referred to as pure constructor CTRSs in the literature [16] and are a subset of 4-CTRSs [22]. They can model first-order functional programs, logic programs, and functional logic programs [5] and are suitable for observing and discussing common problems arising from inversion without considering different language specifications.

The purpose of these systems is to describe relations f from n ground constructor terms to m ground constructor terms, that is,

$$f(s_1, \ldots, s_n) \ \to^*_{\mathcal{R}} \ \langle t_1, \ldots, t_m \rangle.$$

We assume that the signature includes special constructors $\langle\rangle/m$ intended to contain the m output and function symbols of the form `mul`/2/1 and $\texttt{mul}_{\{2\}\{1\}}/2/1$.

Definition 1 (CCS). *A* conditional constructor term rewriting system $\mathcal{R}$, *abbreviated* CCS, *is a CTRS over* $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ *if each rule in* $\mathcal{R}$ *is of the form*

$$l_0 \to r_0 \Leftarrow l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k,$$

where each $l_i \to r_i$ $(0 \leq i \leq k)$ *is of the form* $f^i(p^i_1, \ldots, p^i_{n_i}) \to \langle q^i_1, \ldots, q^i_{m_i} \rangle$ *such that* $f^i/n_i/m_i \in \mathcal{D}$, $\langle\rangle/m_i \in \mathcal{C}$, *and* p^i_j *and* $q^i_{j'}$ *are constructor terms.*

We shall only consider the associated ground constructor-based rewrite relation [20] described in Sect. 3.1. The reductions $f(s_1, \ldots, s_n) \to^*_{\mathcal{R}} \langle t_1, \ldots, t_m \rangle$, where all s_i and t_j are ground constructor terms, specified by a CCS, can only be 1-step reductions, that is, $f(s_1, \ldots, s_n) \to_{\mathcal{R}} \langle t_1, \ldots, t_m \rangle$. The left- and right-hand side of the rules are not unifiable, prohibiting 0-step reductions; there is one function symbol in the initial term, and each rule-application removes exactly one function symbol, prohibiting reductions with more than one step. This also simplifies the correctness proof of the semi-inversion algorithm, which can be proven by induction over the depth of the rewrite steps [22, Def. 7.1.4].

The next example defines a rewrite relation by overlapping rules.

Example 2. This CCS defines a one-to-many rewrite relation `perm`/1/1 between a list and all its permutations, *e.g.*, `perm([1|[2|[]]])` $\to$ `⟨[1|[2|[]]]⟩` and `perm([1|[2|[]]])` $\to$ `⟨[2|[1|[]]]⟩`. It has two defined function symbols `perm`/1/1 and `del`/1/2 and a set of constructors, including two list constructors, `[]`/0 and `[·|·]`/2, and two special output constructors, $\langle\cdot\rangle/1$ and $\langle\cdot,\cdot\rangle/2$. The defined function symbol `perm` depends on `del`, which removes an arbitrary element from a list and returns the removed element and the remaining list. The nondeterministic relation is caused by the overlapping rules `r3` and `r4`.

```
r1: perm([]) → ⟨[]⟩
r2: perm(x) → ⟨[y|z]⟩ ⇐ del(x) → ⟨y,u⟩ ∧ perm(u) → ⟨z⟩
r3: del([x|y]) → ⟨x,y⟩
r4: del([x|y]) → ⟨z, [x|u]⟩ ⇐ del(y) → ⟨z,u⟩
```

A CCS can be nondeterministic by overlapping rules, as in Example 2, and by what we call *extra variables*[2], i.e., variables occurring on the right-hand side r of a rule but neither in its left-hand l side nor in its conditions c, i.e., $\mathcal{V}ar(r) \setminus (\mathcal{V}ar(l) \cup \mathcal{V}ar(c))$. In case a system has no extra variables, we call it *extra-variable free*, abbreviated *EV-free*. EV-free CCSs are a subset of 3-CTRSs [22], and pcDCTRSs [18] are a subset of EV-free CCSs.

[2] In this case, we follow the terminology of [19] —these are not to be confused with "extra variables" as defined by Ohlebusch [22], i.e., $(\mathcal{V}ar(r) \cup \mathcal{V}ar(c)) \setminus \mathcal{V}ar(l)$.

The extra variables cause infinite branching in the ground constructor-based rewrite relation; for example, a rule $f() \to \langle x \rangle$ represents an infinite ground constructor-based rewrite relation $\{f() \to_{\mathcal{R}} \langle a \rangle, f() \to_{\mathcal{R}} \langle b \rangle, \ldots\}$. Intuitively, these variables can be interpreted as *logic variables* subsuming all possible ground constructor terms. Extra variables require efficient implementations that do not naively produce the entire ground constructor-based rewrite relation. *Narrowing* is a well-established rewriting method, where matching is replaced by unification; see, e.g., [5] for further details, [3,12] for a survey, and [19] for the use in partial inversion.

3.3 Semi-inverse

The reader has already seen an example of semi-inversion in Sect. 2. Next, we define semi-inversion formally and illustrate it with examples of the algorithm.

Definition 2 (semi-inverse). *Let $\mathcal{R}$ and $\underline{\mathcal{R}}$ be CCSs over $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ and $\underline{\mathcal{F}} = \underline{\mathcal{C}} \uplus \underline{\mathcal{D}}$, respectively, with $f/n/m \in \mathcal{D}$ and $\underline{f}_{IO}/\underline{n}/\underline{m} \in \underline{\mathcal{D}}$, where $I = \{i_1, \ldots, i_a\}$ and $O = \{o_1, \ldots, o_b\}$ are index sets such that $\underline{n} = a + b$ and $\underline{m} = m + n - \underline{n}$. Then, $\underline{\mathcal{R}}$ is a* semi-inverse of $\mathcal{R}$ *w.r.t. f, I, and O if for all ground constructor terms $s_1, \ldots, s_n$, $t_1, \ldots, t_m \in T(\mathcal{C} \setminus \{\langle\rangle\}, \emptyset)$,*

$$f(s_1, \ldots, s_n) \to_{\mathcal{R}} \langle t_1, \ldots, t_m \rangle \quad \Leftrightarrow$$
$$\underline{f}_{IO}(s_{i_1}, \ldots, s_{i_a}, t_{o_1}, \ldots, t_{o_b}) \to_{\underline{\mathcal{R}}} \langle s_{i_{a+1}}, \ldots, s_{i_n}, t_{o_{b+1}}, \ldots, t_{o_m} \rangle$$

where divisions $\{i_1, \ldots, i_a\} \uplus \{i_{a+1}, \ldots, i_n\} = \{1, \ldots, n\}$ and $\{o_1, \ldots, o_b\} \uplus \{o_{b+1}, \ldots, o_m\} = \{1, \ldots, m\}$. We assume that the name and the parameters of $\underline{f}_{IO}$ are ordered according to $<$-order on the indices.

The reason the semi-inversion algorithm produces a CCS and not an EV-free CCS lies in the nature of full-inversion, i.e., the most specific inversion problem, as demonstrated by the next example.

Example 3. Full inversion of the EV-free CCS $\texttt{fst(x,y)} \to \langle \texttt{x} \rangle$ unavoidably creates a CCS with extra variables, namely, $\underline{\texttt{fst}}_{\emptyset,\{1\}}\texttt{(x)} \to \langle \texttt{x,y} \rangle$.

Sometimes the semi-inverted system and its original system define the same rewrite relation but are defined differently, as in the following examples.

Example 4 (Ex. 2, continued). The semi-inverse of `perm` w.r.t. index sets $I = \emptyset$ and $O = \{1\}$, i.e., a full inversion, is a CCS that defines the same permutation relation by different rules. Here, $\underline{\texttt{del}}_{\emptyset,\{1,2\}}$ inserts an element randomly into a list, whereas the original `del` removes an arbitrary element from the list.

$\underline{\texttt{r1}}\colon \underline{\texttt{perm}}_{\emptyset,\{1\}}\texttt{([])} \to \langle \texttt{[]} \rangle$
$\underline{\texttt{r2}}\colon \underline{\texttt{perm}}_{\emptyset,\{1\}}\texttt{([y|z])} \to \langle \texttt{x} \rangle \Leftarrow \underline{\texttt{perm}}_{\emptyset,\{1\}}\texttt{(z)} \to \langle \texttt{u} \rangle \wedge \underline{\texttt{del}}_{\emptyset,\{1,2\}}\texttt{(y,u)} \to \langle \texttt{x} \rangle$
$\underline{\texttt{r3}}\colon \underline{\texttt{del}}_{\emptyset,\{1,2\}}\texttt{(x,y)} \to \langle \texttt{[x|y]} \rangle$
$\underline{\texttt{r4}}\colon \underline{\texttt{del}}_{\emptyset,\{1,2\}}\texttt{(z,[x|u])} \to \langle \texttt{[x|y]} \rangle \Leftarrow \underline{\texttt{del}}_{\emptyset,\{1,2\}}\texttt{(z,u)} \to \langle \texttt{y} \rangle$

3.4 Modeling Programming Languages and Evaluation Strategies

EV-free CCSs are suitable for modeling logic languages such as Prolog, as seen in the next example, where predicates are modeled by function symbols with co-arity 0. In general, logic programs require narrowing, as we shall see below.

Example 5. The classic predicate append can be modeled by the two rules `app([],y,y)` $\to \langle\rangle$ and `app([h|t],y,[h|z])` $\to \langle\rangle \Leftarrow$ `app(t,y,z)` $\to \langle\rangle$.

The evaluation order of the conditions, i.e., the *strategy*, does not affect the correctness of a rewriting, but the conditions and their order may require narrowing. Instead of describing when there exists an evaluation order, which would only require the faster term rewriting, it is standard to fix the order to be from left to right and to define for which systems there is an order that would only require term rewriting. We follow [16] and define these properties for CCSs, and not only EV-free CCSs as in [22]. For a rule $l \to r \Leftarrow l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k$, a variable x in a condition l_i, i.e., $x \in \mathcal{V}ar(l_i)$, is *known* if $x \in \mathcal{V}ar(l, r_1, \ldots, r_{i-1})$, and *unknown* otherwise. The rule is *left-to-right deterministic*[3] if all variables on the left-hand sides of the conditions are known, i.e., $\mathcal{V}ar(l_i) \subseteq \mathcal{V}ar(l, r_1, \ldots, r_{i-1})$, and a CCS is *left-to-right deterministic* if all its rules are left-to-right deterministic.

A left-to-right deterministic and EV-free CCS does *not* require narrowing, and it is desirable for a semi-inverter to produce such systems [22, Sect. 7.2.5]. In addition, these systems provide a good basis for modeling functional programs [5]. However, other requirements include *orthogonal* rules, i.e., non-overlapping rules and left-linearity. These requirements will not be a part of our design focus for the semi-inversion algorithm, but we will comment on where to check for such paradigm-specific properties in the algorithm in Sect. 4.

Moreover, an EV-free CCS can model reversible languages by ensuring right-orthogonality and non-deletion. Nishida et al. [18] performed a reversibilization of a possibly irreversible pcDCTRS[4] by labeling each right-hand side of a rule with a unique constructor, i.e., right-orthogonality, and recording all deleted values in a trace, i.e., non-deletion. Thus, their resulting pcDCTRSs are reversible.

4 The Semi-inversion Algorithm

The polyvariant semi-inverter is presented in a modular way, including the local inversion and a heuristic to improve the semi-inversion by reordering the conditions. The algorithm semi-inverts a CCS w.r.t. a given function symbol and a pair of input-output index sets into a new CCS. It terminates and yields correct semi-inverse systems, as shown at the end of this section.

[3] Left-to-right determinism is referred to as "determinism" in term rewriting literature, but we make a rather clear distinction between this property, deterministic computations, and deterministic input-output relations, i.e., functions.

[4] Equivalent to left-to-right deterministic EV-free CCSs with orthogonal rules.

The semi-inverter labels all function symbols with two index sets, I and O, that contain the indices of the known terms of the left-hand and right-hand sides of a rule, respectively, and locally inverts every rule after reordering the conditions such that all known variables given by the index sets occur on the left-hand side and the rest occur on the right-hand side of the new rule. The control of rule generation, the heuristic and indices O extend the partial inverter [19].

4.1 Control of Rule Generation

The recursive semi-inversion algorithm (Fig. 3) controls the local inversion (Fig. 4) of the conditional rewrite rules. Given a rewrite system $\mathcal{R}$, an initial function symbol f and the initial input-output index sets I and O, the algorithm produces the semi-inverse rewrite system $\underline{\mathcal{R}}_{\underline{f}_{IO}}$. It keeps track of the function symbols that have been semi-inverted (in set `Done`) and those that are pending semi-inversions (in set `Pend`) to address circular dependencies between rules.

A pending task $(f,I,O) \in$ `Pend` is selected, and each of the rules defining f in $\mathcal{R}$ is semi-inverted, which may lead to new semi-inversion tasks. The auxiliary procedure `getdep` collects all function symbols and their input-output index sets on which the conditions of a set of inverted rules depend. This procedure helps determine new reachable tasks after semi-inverting the rules. Using reachability for semi-inversion reduces the risk of exponentially increasing the size of $\underline{\mathcal{R}}_{\underline{f}_{IO}}$; semi-inversion is *polyvariant inversion* of $\mathcal{R}$ in that it may produce several semi-inversions of the same function symbol, namely, one for each input-output index set. Eventually, all reachable semi-inverses are generated then no pending task exist and the algorithm returns the self-contained semi-inverted system $\underline{\mathcal{R}}_{\underline{f}_{IO}}$.

At this point, as an add-on, the type of the new rewrite system can be syntactically checked. For example, if none of the semi-inverted rules contains an extra variable, then the system is marked as EV-free, and if all function symbols in the CSS are defined by orthogonal rules, then this system corresponds to a first-order

```
seminv(Pend, Done) =
  if Pend = ∅ then ∅ else
  // choose a pending task for semi-inversion
  (f,I,O) ∈ Pend;
  // semi-invert all rules of f with index sets I and O
  f-Rules_original := { ρ | ρ : l → r ⇐ c ∈ R, root(l) = f };
  f-Rules_inverted := { localinv(ρ,I,O) | ρ ∈ f-Rules_original };
  // update the pending and done sets
  NewDep := getdep(f-Rules_inverted) \ Done;
  f-Rules_inverted ∪ seminv((Pend ∪ NewDep) \ {(f,I,O)}, Done ∪ {(f,I,O)});
getdep(Rules) =
  { (f,I,O) | l → r ⇐ c ∈ Rules, l_i → r_i ∈ c, root(l_i) = f_IO };
```

Fig. 3. Recursive semi-inversion algorithm.

functional program. This is the strength of using conditional term rewriting systems as a foundation for studying semi-inversion: they smoothly model inversion problems across a range of different important programming paradigms.

The algorithm *terminates* for any (f,I,O) because the numbers of function symbols and their possible index sets are finite for any given $\mathcal{R}$. In each recursion, a task is semi-inverted and moved from `Pend` to `Done`. Eventually, no more tasks can be added to `Pend` that are not already in `Done`, and the algorithm terminates.

Invocation of the semi-inverter in Fig. 3 is done by $\mathsf{seminv}(\{(f,I,O)\},\ \emptyset)_{\mathcal{R}}$, where the read-only $\mathcal{R}$ is global for the sake of simplicity. A new system $\mathcal{R}_{\underline{f}_{IO}}$ with all semi-inverted functions *reachable* from the initial task (f,I,O) is returned.

Definition 3 (Semi-inverter). *Given a CCS $\mathcal{R}$, a defined function symbol $f/n/m \in \mathcal{D}$, and two index sets $I \subseteq \{1,\ldots,n\}$ and $O \subseteq \{1,\ldots,m\}$, the semi-inverter in Fig. 3 yields the CCS*

$$\mathcal{R}_{\underline{f}_{IO}} = \mathsf{seminv}(\{(f,I,O)\},\emptyset)_{\mathcal{R}}.$$

Note that if the initial pending set contains two or more tasks, they are semi-inverted together by seminv, which may be useful for practical reasons.

$$\mathcal{R}_{\underline{f}_{IO}} \cup \mathcal{R}_{\underline{g}_{I'O'}} = \mathsf{seminv}(\{(f,I,O)\} \cup \{(g,I',O')\},\emptyset)_{\mathcal{R}}.$$

4.2 Local Semi-inversion of Conditional Rules

The form of the rules with a rule head followed by a sequence of flat conditions considerably simplifies the local inversion. Given the index sets I and O, a rule

$$(l \to r \Leftarrow l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k)$$

is locally semi-inverted into a *left-to-right deterministic* rule

$$(l' \to r' \Leftarrow l'_1 \to r'_1 \wedge \ldots \wedge l'_k \to r'_k),$$

i.e., satisfying $\mathcal{V}ar(l', r'_1, \ldots, r'_{i-1}) \supseteq \mathcal{V}ar(l'_i)$ for all $1 \leq i \leq k$.

Local inversion (Fig. 4) first generates the new head $l' \to r'$ by rearranging the terms of l and r according to I and O as required for semi-inversion (Def. 2). After $\mathsf{heuristic}$ reorders the conditions, $\mathsf{localinvc}$ ensures that the new rule is left-to-right deterministic by transforming the conditions $l_1 \to r_1 \wedge \ldots \wedge l_k \to r_k$ from left to right such that all terms in the ith condition $(l_i \to r_i)$ that depend only on the already *known variables* $\mathcal{V}ar(l, r_1, \ldots, r_{i-1})$ are moved to the new left-hand side l'_i and all other terms are moved to the new right-hand side r'_i (ordered by increasing index). Therefore, l'_i contains terms that depend on known variables (including ground terms), and r'_i contains all the other terms. At the same time, the function symbol $\underline{f_i}$ at the root of l'_i is labeled with the corresponding input-output sets. This transformation is repeated recursively from left to right for each condition while updating the set of known variables. In this way, the semi-inverted rule becomes left-to-right deterministic, and each new condition $l'_i \to r'_i$ uses the maximum number of known terms in l'_i. This step is not necessary for correctness but reduces the search space of the intended reduction strategy.

```
localinv(f(p1,...,pn) → ⟨q1,...,qm⟩ ⇐ c, {i1,...,ia}, {o1,...,ob}) =
  // semi-invert the rule head and label the function symbol
  {ia+1,..., in} := {1,...,n} \ {i1,...,ia}
  {ob+1,..., om} := {1,...,m} \ {o1,...,ob}
  lhs := f{i1,...,ia}{o1,...,ob}(pi1,...,pia,qo1,...,qob)
  rhs := ⟨pia+1,...,pin,qob+1,...,qom⟩
  // locally invert the conditions after reordering
  Var := Var(lhs)
  c'  := heuristic(c, Var)      // reorder conditions of rule
  c'' := localinvc(c',Var)      // local inversion of conditions
  // return the inverted rule
  lhs → rhs ⇐ c''
localinvc(c, Var) = case c of
  // if no condition, then return the empty condition
  ϵ  =>  ϵ
  // else invert the left-most condition
  f(p1,...,pn) → ⟨q1,...,qm⟩ ∧ Restc  =>
    // build the index sets
    {i1,  ..., ia} := {i | i ∈ {1,...,n}, Var(pi) ⊆ Var}
    {ia+1,..., in} := {1,...,n} \ {i1,...,ia}
    {o1,  ..., ob} := {o | o ∈ {1,...,m}, Var(qo) ⊆ Var}
    {ob+1,..., om} := {1,...,m} \ {o1,...,ob}
    // locally invert and label the left-most condition
    lhs := f{i1,...,ia}{o1,...,ob}(pi1,...,pia,qo1,...,qob)
    rhs := ⟨pia+1,...,pin,qob+1,...,qom⟩
    // return the inverted conditions
    lhs → rhs ∧ localinvc(Restc, Var ∪ Var(rhs))
```

Fig. 4. Local inversion of a conditional rule.

4.3 A Heuristic Approach to Reordering Conditions

We have chosen a greedy heuristic to reorder the conditions in a rule before semi-inverting them, which works surprisingly well. The procedure heuristic (shown in Fig. 5) reorders the conditions according to the *percentages of known parameters* such that the condition with the highest percentage comes first. This procedure dynamically updates the set of known variables each time a condition is moved to the head of the sequence and recursively applies the reordering to the remaining conditions. Clearly, different sets of known variables can lead to different orders of the conditions. The intention with this heuristic is to syntactically exploit as much known information as possible without having to rely on an extra analysis.

Other reordering methods could be used instead. The algorithm by Mogensen [14], which semi-inverts non-overlapping rules in a guarded equational language without extra variables, searches through *all possible semi-inversions* and uses additional semantic information about primitive operators. An exhaustive search will find better orders than a local heuristic, but the search will take more time. This is the familiar trade-off between the accuracy and run time of

```
heuristic(c, KnownVar) = case c of
  // if no condition, then return the empty condition
  ϵ  =>  ϵ
  // else find condition with highest percentage of known parameters
  l1 → r1 ∧ ...∧ lk → rk  =>
    // determine percentages
    Percent1 := percent(l1 → r1, KnownVar)
      ...      ...
    Percentk := percent(lk → rk, KnownVar)
    (Pi,i)     := maxPercent((Percent1,1),...,(Percentk,k))
    // select condition, reorder remaining conditions in updated variable set
    li → ri ∧ heuristic(c \ (li → ri), KnownVar ∪ Var(li,ri))

percent(f(p1, ..., pn) → ⟨q1, ..., qm⟩, KnownVar) =
  // determine index sets of known parameters
  I := {i | i ∈ {1, ..., n}, Var(pi) ⊆ KnownVar }
  O := {j | j ∈ {1, ..., m}, Var(qj) ⊆ KnownVar }
  // return percentage of known parameters
  (|I| + |O|) / (m + n)        // known = |I| + |O|, total = m + n
```

Fig. 5. A greedy heuristic for reordering conditions.

a program analysis. The heuristic always finds a reordering (perhaps leading to extra variables), while the semi-inverter [14] may halt with no answer due to the limitations of the language —a later inverter [15] allows functional parameters.

There is no fixed order of conditions that avoids extra variables for all possible semi-inversions of a given rewrite system. Our experiments show that the heuristic usually improves the resulting semi-inversion, but it can also be deceived, as shown in the following example.

Example 6. Given a system consisting of `r1`–`r3`, the semi-inversion of `test` w.r.t. $I = \{1\}$ and $O = \{2\}$ yields the system `s1`–`s3`, which has an extra variable in function $\underline{\texttt{fst}}_{\{1\}\{1\}}$, whereas one produced without the heuristic would be EV-free.

```
r1: test(x,y) → ⟨w,z⟩    ⇐ copy(x,y) → ⟨w,z⟩ ∧ fst(x,y) → ⟨z⟩
r2: fst(x,y) → ⟨x⟩          r3: copy(x,y) → ⟨x,y⟩

s1: test{1}{2}(x,z) → ⟨y,w⟩ ⇐ fst{1}{1}(x,z) → ⟨y⟩ ∧ copy{1,2}{2}(x,y,z) → ⟨w⟩
s2: fst{1}{1}(x,x) → ⟨y⟩      s3: copy{1,2}{2}(x,y,y) → ⟨x⟩
```

4.4 Correctness of the Semi-inversion Algorithm

The correctness of the semi-inversion algorithm is proven by first defining $\underline{\mathcal{R}}_{all}$ by semi-inverting all possible semi-inversion tasks for the function symbols in $\mathcal{R}$.

Definition 4 ($\underline{\mathcal{R}}_{all}$). *Let $\mathcal{R}$ be a CCS, and let $P = \{(f, I, O) \mid f/n/m \in \mathcal{D}, I \subseteq \{1, \ldots, n\}, O \subseteq \{1, \ldots, m\}\}$ be the pending set consisting of all semi-inversion tasks of all function symbols defined in $\mathcal{R}$. Then, we define*

$$\underline{\mathcal{R}}_{all} = \mathsf{seminv}(P, \emptyset)_{\mathcal{R}}.$$

The following theorem can be proven in two steps: First, the rewrite steps of f are in $\mathcal{R}$ if and only if the rewrite steps of its semi-inverse $\underline{f}_{IO}$ are in $\underline{\mathcal{R}}_{all}$, and secondly, the rewrite steps of $\underline{f}_{IO}$ are in $\underline{\mathcal{R}}_{all}$ if and only if they are in $\underline{\mathcal{R}}_{\underline{f}_{IO}}$. The proofs are omitted due to lack of space.

Theorem 1. *Let $\mathcal{R}$ be a CCS, and let $\underline{\mathcal{R}}_{\underline{f}_{IO}} = \mathsf{seminv}(\{(f, I, O)\}, \emptyset)_{\mathcal{R}}$ for a function symbol $f/n/m \in \mathcal{D}$ and index sets $I \subseteq \{1, \ldots, n\}$ and $O \subseteq \{1, \ldots, m\}$. Then, $\underline{\mathcal{R}}_{\underline{f}_{IO}}$ is a semi-inverse of $\mathcal{R}$ w.r.t. f, I, and O.*

5 Application of the Semi-inverter

The semi-inverter has been fully implemented (in Prolog), and in the following, we will present the results: a series of semi-inversions of a discrete simulation of a free fall, each solving a different problem, a decrypter from a simple encrypter, and the inversion of an inverter for a reversible programming language.

5.1 Discrete Simulation of a Free Fall

We consider a discrete simulation of an object that falls through a vacuum [4] and use semi-inversion to generate four new programs, each solving a different aspect. The simulation is defined by the equations $v_t = v_{t-1} + g$ and $h_t = h_{t-1} - v_t + g/2$, where $g \approx 10\,m/s^2$ is the approximate gravitational acceleration. The following system $\mathtt{fall}_0$ yields the object's velocity v_t ($\mathtt{v}$) and height h_t ($\mathtt{h}$) at time t, given t ($\mathtt{t}$) and initial velocity v_0 ($\mathtt{v}_0$) and height h_0 ($\mathtt{h}_0$).

$$\mathtt{fall}_0(\mathtt{v},\mathtt{h},0) \;\rightarrow\; \langle \mathtt{v},\mathtt{h}\rangle$$

$$\mathtt{fall}_0(\mathtt{v}_0,\mathtt{h}_0,\mathtt{s}(\mathtt{t})) \rightarrow \langle \mathtt{v},\mathtt{h}\rangle \Leftarrow \mathtt{add}(\mathtt{v}_0, \mathtt{s}^5(0)) \rightarrow \langle \mathtt{v}_n\rangle \wedge \mathtt{height}(\mathtt{h}_0,\mathtt{v}_n) \rightarrow \langle \mathtt{h}_n\rangle \wedge \mathtt{fall}_0(\mathtt{v}_n,\mathtt{h}_n,\mathtt{t}) \rightarrow \langle \mathtt{v},\mathtt{h}\rangle$$

$$\mathtt{height}(\mathtt{h}_0,\mathtt{v}_n) \rightarrow \langle \mathtt{h}_n\rangle \;\Leftarrow\; \mathtt{add}(\mathtt{h}_0,\mathtt{s}^5(0)) \rightarrow \langle \mathtt{h}_{temp}\rangle \wedge\; \mathtt{sub}(\mathtt{h}_{temp},\mathtt{v}_n) \rightarrow \langle \mathtt{h}_n\rangle$$

Original	$\mathtt{fall}_0\colon\ (\mathtt{v}_0, \mathtt{h}_0, \mathtt{t}) \rightarrow (\mathtt{v}, \mathtt{h})$
Full inv.	$\mathtt{fall}_1\colon\ (\mathtt{v}, \mathtt{h}) \rightarrow (\mathtt{v}_0, \mathtt{h}_0, \mathtt{t})$
Partial inv.	$\mathtt{fall}_2\colon\ (\mathtt{t}, \mathtt{v}, \mathtt{h}) \rightarrow (\mathtt{v}_0, \mathtt{h}_0)$
Semi-inv. #1	$\mathtt{fall}_3\colon\ (\mathtt{v}_0, \mathtt{t}, \mathtt{h}) \rightarrow (\mathtt{h}_0, \mathtt{v})$
Semi-inv. #2	$\mathtt{fall}_4\colon\ (\mathtt{v}_0, \mathtt{t}, \mathtt{v}) \rightarrow (\mathtt{h}_0, \mathtt{h})$

The system fall_0 is geared towards solving the 'forward' problem, while finding a solution to the 'backward' problem of determining the origin of a fall may be equally interesting, it requires a new set of rules. Full inversion algorithms can transform fall_0 into a new system fall_1 solving the backward problem, while other problems require partial or semi-inversion.

These four programs $\mathtt{fall}_1$ to $\mathtt{fall}_4$ are successfully generated by the semi-inversion algorithm as shown in Fig. 6 (dependency functions are omitted for clarity). The difference between the order of the conditions in the four inversions indicates that the heuristic has taken action.

5.2 Encrypter and Decrypter

The automatic generation of decrypters is fascinating. The following symmetric encrypter is a modification of a simple encryption method suggested by

$$
\begin{array}{l|l}
\texttt{fall}_1(\texttt{v},\texttt{h}) \to \langle \texttt{v},\texttt{h},0\rangle & \texttt{fall}_2(0,\texttt{v},\texttt{h}) \to \langle \texttt{v},\texttt{h}\rangle \\
\texttt{fall}_1(\texttt{v},\texttt{h}) \to \langle \texttt{v}_0,\texttt{h}_0,\texttt{s(t)}\rangle \Leftarrow & \texttt{fall}_2(\texttt{s(t)},\texttt{v},\texttt{h}) \to \langle \texttt{v}_0,\texttt{h}_0\rangle \Leftarrow \\
\quad \texttt{fall}_1(\texttt{v},\texttt{h}) \to \langle \texttt{v}_n,\texttt{h}_n,\texttt{t}\rangle \wedge & \quad \texttt{fall}_2(\texttt{t},\texttt{v},\texttt{h}) \to \langle \texttt{v}_n,\texttt{h}_n\rangle \wedge \\
\quad \underline{\texttt{add}}_{\{2\}\{1\}}(\texttt{s}^{10}\texttt{(0)},\texttt{v}_n) \to \langle \texttt{v}_0\rangle \wedge & \quad \underline{\texttt{add}}_{\{2\}\{1\}}(\texttt{s}^{10}\texttt{(0)},\texttt{v}_n) \to \langle \texttt{v}_0\rangle \wedge \\
\quad \underline{\texttt{height}}_{\{2\}\{1\}}(\texttt{v}_n,\texttt{h}_n) \to \langle \texttt{h}_0\rangle & \quad \underline{\texttt{height}}_{\{2\}\{1\}}(\texttt{v}_n,\texttt{h}_n) \to \langle \texttt{h}_0\rangle \\
\hline
\texttt{fall}_3(\texttt{v},0,\texttt{h}) \to \langle \texttt{h},\texttt{v}\rangle & \texttt{fall}_4(\texttt{v},0,\texttt{v}) \to \langle \texttt{h},\texttt{h}\rangle \\
\texttt{fall}_3(\texttt{v}_0,\texttt{s(t)},\texttt{h}) \to \langle \texttt{h}_0,\texttt{v}\rangle \Leftarrow & \texttt{fall}_4(\texttt{v}_0,\texttt{s(t)},\texttt{v}) \to \langle \texttt{h}_0,\texttt{h}\rangle \Leftarrow \\
\quad \texttt{add}(\texttt{v}_0,\texttt{s}^{10}\texttt{(0)}) \to \langle \texttt{v}_n\rangle \wedge & \quad \texttt{add}(\texttt{v}_0,\texttt{s}^{10}\texttt{(0)}) \to \langle \texttt{v}_n\rangle \wedge \\
\quad \texttt{fall}_3(\texttt{v}_n,\texttt{t},\texttt{h}) \to \langle \texttt{h}_n,\texttt{v}\rangle \wedge & \quad \texttt{fall}_4(\texttt{v}_n,\texttt{t},\texttt{v}) \to \langle \texttt{h}_n,\texttt{h}\rangle \wedge \\
\quad \underline{\texttt{height}}_{\{2\}\{1\}}(\texttt{v}_n,\texttt{h}_n) \to \langle \texttt{h}_0\rangle & \quad \underline{\texttt{height}}_{\{2\}\{1\}}(\texttt{v}_n,\texttt{h}_n) \to \langle \texttt{h}_0\rangle
\end{array}
$$

Fig. 6. The $\texttt{fall}_0$, its full inversion $\texttt{fall}_1$, the partial inversion $\texttt{fall}_2$ and the two semi-inversions $\texttt{fall}_3$ and $\texttt{fall}_4$.

Mogensen. It produces an encrypted text `z:zs` given a text formed as an integer list `x:xs` and a key `key`. Encryption cleans the key by mod 4, adds the new value to the first character, and repeats the process recursively for the rest of the text (the modification is that we use mod 4 instead of mod 256, as mod 256 consists of 257 rules). The encrypter is given in Fig. 7 (on the left), and the decrypter is a partial inversion of it with respect to $I = \{2\}$ and $O = \{1\}$, as shown in Fig. 7 (on the right). The decrypter ($\underline{\texttt{encrypt}}_{\{2\}\{1\}}$) produces the decrypted text `x:xs` given the key `key` and the encrypted text `z:zs`. Note that `mod4` is equivalent to $\underline{\texttt{mod4}}_{\{1\}\emptyset}$.

Original encrypter:	Generated decrypter:
`encrypt(nil,key)` $\to \langle$`nil`$\rangle$	$\underline{\texttt{encrypt}}_{\{2\}\{1\}}$`(key,nil)` $\to \langle$`nil`$\rangle$
`encrypt(x:xs,key)` $\to \langle$`z:zs`$\rangle \Leftarrow$	$\underline{\texttt{encrypt}}_{\{2\}\{1\}}$`(key,z:zs)` $\to \langle$`x:xs`$\rangle \Leftarrow$
`mod4(key)` $\to \langle$`y`$\rangle$,	$\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(key)` $\to \langle$`y`$\rangle$,
`add(x,y)` $\to \langle$`z`$\rangle$,	$\underline{\texttt{add}}_{\{2\}\{1\}}$`(y,z)` $\to \langle$`x`$\rangle$,
`encrypt(xs,key)` $\to \langle$`zs`$\rangle$	$\underline{\texttt{encrypt}}_{\{2\}\{1\}}$`(key,zs)` $\to \langle$`xs`$\rangle$
`mod4(0)` $\to \langle$`0`$\rangle$	$\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(0)` $\to \langle$`0`$\rangle$
`mod4(s(0))` $\to \langle$`s(0)`$\rangle$	$\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(s(0))` $\to \langle$`s(0)`$\rangle$
`mod4(s`^{2}`(0))` $\to \langle$`s`^{2}`(0)`$\rangle$	$\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(s`^{2}`(0))` $\to \langle$`s`^{2}`(0)`$\rangle$
`mod4(s`^{3}`(0))` $\to \langle$`s`^{3}`(0)`$\rangle$	$\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(s`^{3}`(0))` $\to \langle$`s`^{3}`(0)`$\rangle$
`mod4(s`^{4}`(x))` $\to \langle$`w0`$\rangle \Leftarrow$ `mod4(x)` $\to \langle$`w0`$\rangle$	$\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(s`^{4}`(x))` $\to \langle$`w0`$\rangle \Leftarrow$ $\underline{\texttt{mod4}}_{\{1\}\emptyset}$`(x)` $\to \langle$`w0`$\rangle$

Fig. 7. Inversion of a simple symmetric encrypter into a decrypter.

5.3 Inverted Inverter

The Janus language is a reversible language [23] where functions can be both called (executed in a forward direction) and uncalled (executed in a backward direction). An inverter for the Janus language creates procedures that are equal

to uncalling the original procedure. Since Janus is a reversible language, the full inversion of such a Janus-inverter is equivalent to itself. A Janus-inverter can be described as a left-to-right deterministic EV-free CCS, and in Fig. 8 (on the left), we have given such an inverter for a subset of the language.

A full inversion of the inverter for the reversible Janus language is equivalent to itself. The result of the semi-inversion algorithm is the Janus inverter shown in Fig. 8 (on the right). If we assume that `invName` and its full inversion `invName` are equivalent, then the produced and original rules are equivalent up to the variable naming and order of the parameters.

Janus-Inverter	Fully inverted Janus-inverter
`inv(proc(name,progr)) → ⟨proc(u,v)⟩ ⇐` `  invName(name) → ⟨u⟩ ∧ inv(progr) → ⟨v⟩`	`inv(proc(u,v)) → ⟨proc(name,progr)⟩ ⇐` `  invName(u) → ⟨name⟩ ∧ inv(v) → ⟨progr⟩`

`inv(+=(x,y)) → ⟨-=(x,y)⟩`
`inv(-=(x,y)) → ⟨+=(x,y)⟩`
`inv(<=>(x,y)) → ⟨<=>(x,y)⟩`
	`inv(-=(x,y)) → ⟨+=(x,y)⟩`
	`inv(+=(x,y)) → ⟨-=(x,y)⟩`
	`inv(<=>(x,y)) → ⟨<=>(x,y)⟩`

`inv(if(x1,y1,y2,x2)) → ⟨if(x2,z1,z2,x1)⟩ ⇐`
`  inv(y1) → ⟨z1⟩∧ inv(y2) → ⟨z2⟩`
`inv(loop(x1,y1,y2,x2)) → ⟨loop(x2,z1,v2,x1)⟩ ⇐`
`  inv(y1) → ⟨z1⟩ ∧ inv(y2) → ⟨z2⟩`
	`inv(if(x2,z1,z2,x1)) → ⟨if(x1,y1,y2,x2)⟩ ⇐`
	`  inv(z1) → ⟨y1⟩ ∧ inv(z2) → ⟨y2⟩`
	`inv(loop(x2,z1,z2,x1)) → ⟨loop(x1,y1,y2,x2)⟩ ⇐`
	`  inv(z1) → ⟨y1⟩ ∧ inv(z2) → ⟨y2⟩`

`inv(call(name)) → ⟨call(u)⟩ ⇐`
`  invName(name) → ⟨u⟩`
`inv(uncall(name)) → ⟨uncall(u)⟩ ⇐`
`  invName(name) → ⟨u⟩`
	`inv(call(u)) → ⟨call(name)⟩ ⇐`
	`  invName(u) → ⟨name⟩`
	`inv(uncall(u)) → ⟨uncall(name)⟩ ⇐`
	`  invName(u) → ⟨name⟩`

`inv(sequence(x,y)) → ⟨sequence(u,v)⟩ ⇐`
`  inv(y) → ⟨u⟩ ∧ inv(x) → ⟨v⟩`
`inv(skip) → ⟨skip⟩`
	`inv(sequence(u,v)) → ⟨sequence(x,y)⟩ ⇐`
	`  inv(u) → ⟨y⟩ ∧ inv(v) → ⟨x⟩`
	`inv(skip) → ⟨skip⟩`

Fig. 8. The Janus-inverter `inv` expressed as CCS rules, and its full inverse `inv`.

6 Conclusion

We have shown that polyvariant semi-inversion can be performed for conditional constructor systems (CCSs) that are highly useful for modeling several language paradigms, including logic, functional, and reversible languages. Notably, we have shown that local inversion and a straightforward heuristics suffice for performing the general form of inversion with interesting results. This approach can be used for transformation problems ranging from the inversion of a simple encryption algorithm or a physical discrete-event simulation to a program inverter for a reversible language. We have also implemented the algorithm and shown its correctness. The algorithm makes use of a simple syntactic heuristic that produces good results in our experiments. In full inversion, some auxiliary functions may be partially inverted [17, p. 145], that is, their inversion may also benefit from the algorithm in this paper. Furthermore, the structure of the algorithm is modular such that the heuristic can be easily replaced. In regard to future work, it could be interesting to vary the heuristics with respect to

the type of inversion considering that reversing conditions suffices for full inversion; e.g., by parameterization, we might capture other inversion algorithms in this framework.

Acknowledgment. The authors wish to thank German Vidal and the anonymous reviewers. Support by the EU COST Action IC1405 is acknowledged.

References

1. Abramov, S.M., Glück, R.: The universal resolving algorithm and its correctness: inverse computation in a functional language. Sci. Comput. Program. **43**(2–3), 193–229 (2002)
2. Almendros-Jiménez, J.M., Vidal, G.: Automatic partial inversion of inductively sequential functions. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 253–270. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74130-5_15
3. Antoy, S.: Programming with narrowing: a tutorial. J. Symb. Comput. **45**(5), 501–522 (2010). (version: June 2017, update)
4. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible machine code and its abstract processor architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74510-5_9
5. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
6. Dijkstra, E.W.: Program inversion. In: Bauer, F.L., et al. (eds.) Program Construction. LNCS, vol. 69, pp. 54–57. Springer, Heidelberg (1979). https://doi.org/10.1007/BFb0014657
7. Eppstein, D.: A heuristic approach to program inversion. In: Joshi, A.K. (ed.) IJCAI-85. Proceedings, vol. 1, pp. 219–221. Morgan Kaufmann Inc (1985)
8. Glück, R., Kawabe, M.: A program inverter for a functional language with equality and constructors. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 246–264. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40018-9_17
9. Glück, R., Kawabe, M.: A method for automatic program inversion based on LR(0) parsing. Fundamenta Informaticae **66**, 367–395 (2005)
10. Glück, R., Kawabe, M.: Revisiting an automatic program inverter for Lisp. SIGPLAN Notices **40**(5), 8–17 (2005)
11. Glück, R., Turchin, V.F.: Application of metasystem transition to function inversion and transformation. In: Proceedings of the ISSAC, pp. 286–287. ACM Press (1990)
12. Hanus, M.: The integration of functions into logic programming: from theory to practice. J. Logic Program. **19–20**(Suppl. 1), 583–628 (1994)
13. McCarthy, J.: The inversion of functions defined by Turing machines. In: Shannon, C., McCarthy, J. (eds.) Automata Studies, pp. 177–181. Princeton University Press, Princeton (1956)
14. Mogensen, T.Æ.: Semi-inversion of guarded equations. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 189–204. Springer, Heidelberg (2005). https://doi.org/10.1007/11561347_14
15. Mogensen, T. Æ.: Semi-inversion of functional parameters. In: Proceedings of the PEPM, pp. 21–29. ACM (2008)

16. Nagashima, M., Sakai, M., Sakabe, T.: Determinization of conditional term rewriting systems. Theor. Comput. Sci. **464**, 72–89 (2012)
17. Nishida, N.: Transformational approach to inverse computation in term rewriting. Ph.D. thesis, Graduate School of Engineering, Nagoya University (2004)
18. Nishida, N., Palacios, A., Vidal, G.: Reversible computation in term rewriting. J. Logic. Algebr. Methods Program. **94**, 128–149 (2018)
19. Nishida, N., Sakai, M., Sakabe, T.: Partial inversion of constructor term rewriting systems. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 264–278. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_20
20. Nishida, N., Vidal, G.: Program inversion for tail recursive functions. In: Proceedings RTA, LIPIcs, vol. 10, pp. 283–298. Schloss Dagstuhl (2011)
21. Ohlebusch, E.: Transforming conditional rewrite systems with extra variables into unconditional systems. In: Ganzinger, H., McAllester, D., Voronkov, A. (eds.) LPAR 1999. LNCS (LNAI), vol. 1705, pp. 111–130. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48242-3_8
22. Ohlebusch, E.: Advanced Topics in Term Rewriting. Springer, New York (2002). https://doi.org/10.1007/978-1-4757-3661-8
23. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Proceedings of the PEPM, pp. 144–153. ACM (2007)