

## Automatic Partial Inversion of Inductively Sequential Functions<sup>\*</sup>

Jesús M. Almendros-Jiménez<sup>1</sup> and Germán Vidal<sup>2</sup>

<sup>1</sup> University of Almería, Spain. Email: jalmen@ual.es

<sup>2</sup> Technical University of Valencia, Spain. Email: gvidal@dsic.upv.es

**Abstract.** We introduce a new partial inversion technique for first-order functional programs. Our technique is simple, fully automatic, and (when it succeeds) returns a program that belongs to the same class of the original program, namely the class of inductively sequential programs (i.e., typical functional programs). Therefore, it forms an appropriate basis for developing a practically applicable transformation.

### 1 Motivation

Program inversion is a fundamental transformation within the functional programming paradigm. Having a fully automatic inversion tool would be useful because there are many functions that can be seen as the inverse of other, sometimes easier, functions (e.g., product and division, encoding and decoding, compression and decompression, etc).

Given a function  $f$  such that  $f(t_1, \dots, t_n) = t$ , the *total inversion* of  $f$  is a new function  $f^{-1}$  such that  $f^{-1}(t) = \langle t_1, \dots, t_n \rangle$  for all terms  $t_1, \dots, t_n, t$ . In this work, and in contrast to most of the work on program inversion, we consider the computation of *partial inverses*. Given a function  $f$  such that  $f(t_1, \dots, t_n) = t$ , the *partial inversion* of  $f$  w.r.t. the set of parameters  $I = \{i_1, \dots, i_m\} \subset \{1, \dots, n\}$  is a new function  $\bar{f}_I$  such that  $\bar{f}_I(t, t_{i_1}, \dots, t_{i_m}) = \langle t_{j_1}, \dots, t_{j_k} \rangle$  for all terms  $t_1, \dots, t_n, t$ , with  $\{j_1, \dots, j_k\} = \{1, \dots, n\} \setminus I$ . Clearly, partial inversion subsumes total inversion (i.e., when  $I = \emptyset$ ).

Consider, for instance, the usual definition of the addition on natural numbers (built from *zero* and *succ*):

$$\begin{aligned} \text{add}(\text{zero}, y) &\rightarrow y \\ \text{add}(\text{succ}(x), y) &\rightarrow \text{succ}(\text{add}(x, y)) \end{aligned}$$

Here, there exist three possible partial inverses:  $\text{add}_{\emptyset}$  (the total inversion),  $\text{add}_{\{1\}}$  and  $\text{add}_{\{2\}}$ . The specifications of these partial inverses are as follows:

$$\begin{aligned} \overline{\text{add}}_{\emptyset}(t) = \langle t_1, t_2 \rangle &\Leftrightarrow \text{add}(t_1, t_2) = t \\ \overline{\text{add}}_{\{1\}}(t, t_1) = t_2 &\Leftrightarrow \text{add}(t_1, t_2) = t \\ \overline{\text{add}}_{\{2\}}(t, t_2) = t_1 &\Leftrightarrow \text{add}(t_1, t_2) = t \end{aligned}$$

<sup>\*</sup> This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02 and by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054.

Their definitions are, respectively, the following:

$$\begin{aligned} \overline{add}_{\emptyset}(y) &\rightarrow \langle zero, y \rangle \\ \overline{add}_{\emptyset}(succ(w)) &\rightarrow \mathbf{let} \langle x, y \rangle = \overline{add}_{\emptyset}(w) \mathbf{in} \langle succ(x), y \rangle \\ \overline{add}_{\{1\}}(y, zero) &\rightarrow y \\ \overline{add}_{\{1\}}(succ(w), succ(x)) &\rightarrow \overline{add}_{\{1\}}(w, x) \\ \overline{add}_{\{2\}}(y, y) &\rightarrow zero \\ \overline{add}_{\{2\}}(succ(w), y) &\rightarrow succ(\overline{add}_{\{2\}}(w, y)) \end{aligned}$$

Observe that the original definition of function *add* is *inductively sequential* [1]; roughly speaking, a function is inductively sequential when its definition is left-linear (i.e., there are no multiple occurrences of the same variable in the left-hand sides) and without overlapping left-hand sides (i.e., with no unifying left-hand sides). However,

- the definition of the partial inverse  $\overline{add}_{\emptyset}$  has overlapping left-hand sides, and
- the definition of the partial inverse  $\overline{add}_{\{2\}}$  is not left-linear.

Therefore, program inversion can generally produce programs which do not belong to the same class of the original programs.

In this work, however, we are only interested in those program transformations that preserve the inductively sequential nature of the original program (the case of function  $\overline{add}_{\{1\}}$ ). Preserving the inductive sequentiality of the original program is an essential property for having a practically applicable transformation, since typical functional programs are usually inductively sequential.

Furthermore, we consider *partial* inverses because they subsume the computation of *total* inverses and because functions need not be injective. Moreover, there are many practical cases where the computation of a partial inverse is more useful; e.g., function  $\overline{add}_{\{1\}}$  implements the subtraction on natural numbers, while the practical use of the total inverse  $\overline{add}_{\emptyset}$  is not so obvious.

The main features of the partial inversion method that we introduce in this paper can be summarized as follows:

- The method proceeds in a stepwise manner: normalization (introduction of let expressions), partial inversion, and removal of let expressions.
- The method is purely static, i.e., no (partial) computations are performed.
- Finally, our method always terminates, either returning an inductively sequential program—defining the partial inversion of a function—or a failure.

The paper is organized as follows. After introducing some preliminaries in the next section, we present in Sect. 3 our method for partial inversion. Finally, Sect. 4 discusses some related work and Sect. 5 concludes.

## 2 Preliminaries

We follow the standard framework of *term rewriting* [3] for developing our results since it suffices to model the first-order component of many functional programming languages.

In this setting, a set of rewrite rules (or oriented equations)  $l \rightarrow r$  such that  $l$  is a nonvariable term and  $r$  is a term is called a *term rewriting system* (TRS for short); terms  $l$  and  $r$  are called the left-hand side and the right-hand side of the rule, respectively. If there are variables in the right-hand side of a rule that do not appear in the corresponding left-hand side, we say that the TRS contains *extra variables*. In this work, we only consider TRSs without extra variables. Given a TRS  $\mathcal{R}$  over a signature  $\mathcal{F}$ , the *defined* symbols  $\mathcal{D}$  are the root symbols of the left-hand sides of the rules and the *constructors* are  $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$ . We restrict ourselves to finite signatures and TRSs. We denote the domain of terms and *constructor terms* by  $\mathcal{T}(\mathcal{F}, \mathcal{V})$  and  $\mathcal{T}(\mathcal{C}, \mathcal{V})$ , respectively, where  $\mathcal{V}$  is a set of variables with  $\mathcal{F} \cap \mathcal{V} = \emptyset$ .

A TRS  $\mathcal{R}$  is *constructor-based* if the left-hand sides of its rules have the form  $f(s_1, \dots, s_n)$  where  $s_i$  are constructor terms, i.e.,  $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ , for all  $i = 1, \dots, n$ . The set of variables appearing in a term  $t$  is denoted by  $\text{Var}(t)$ . A term  $t$  is *linear* if every variable of  $\mathcal{V}$  occurs at most once in  $t$ .  $\mathcal{R}$  is left-linear if  $l$  is linear for all rule  $l \rightarrow r \in \mathcal{R}$ . The *definition* of  $f$  in  $\mathcal{R}$  is the set of rules in  $\mathcal{R}$  whose root symbol in the left-hand side is  $f$ . A function  $f \in \mathcal{D}$  is left-linear if the rules in its definition are left-linear.

The root symbol of a term  $t$  is denoted by  $\text{root}(t)$ . A term  $t$  is *operation-rooted* (resp. *constructor-rooted*) if  $\text{root}(t) \in \mathcal{D}$  (resp.  $\text{root}(t) \in \mathcal{C}$ ). As it is common practice, a *position*  $p$  in a term  $t$  is represented by a sequence of natural numbers, where  $\epsilon$  denotes the root position. Positions are used to address the nodes of a term viewed as a tree:  $t|_p$  denotes the *subterm* of  $t$  at position  $p$  and  $t[s]_p$  denotes the result of *replacing the subterm*  $t|_p$  by the term  $s$ . A term  $t$  is *ground* if  $\text{Var}(t) = \emptyset$ . A *substitution*  $\sigma$  is a mapping  $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$  from variables to terms such that its domain  $\text{Dom}(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$  is finite. The identity substitution is denoted by *id*. We write  $\overline{o_n}$  for the *sequence of syntactic objects*  $o_1, \dots, o_n$ .

Two (possibly renamed) constructor-based rules  $l \rightarrow r$  and  $l' \rightarrow r'$  *overlap* if there exists a unifier  $\sigma$  such that  $\sigma(l) = \sigma(l')$ . A constructor-based, left-linear TRS without overlapping rules is called *orthogonal*. Inductively sequential TRSs [1] are a subclass of constructor-based orthogonal TRSs. Essentially, a TRS is *inductively sequential* when all its operations are defined by rewrite rules that, recursively, make on their arguments a case distinction analogous to a data type (or structural) induction. Inductive sequentiality is not a limiting condition for programming. In fact, the first-order components of many functional and functional logic programs written in, e.g., Haskell, ML or Curry, are inductively sequential. Also, the class of inductively sequential programs provides for optimal computations both in functional and functional logic programming [1, 2]. A precise definition of inductively sequential TRSs is not necessary in this work

and, thus, we only illustrate this notion with an example (a detailed definition can be found in [1]):

*Example 1.* Consider the following definition of the less-or-equal relation:

$$\begin{aligned} \text{zero} \leq y &\rightarrow \text{true} \\ \text{succ}(x) \leq \text{zero} &\rightarrow \text{false} \\ \text{succ}(x) \leq \text{succ}(y) &\rightarrow x \leq y \end{aligned}$$

This function is inductively sequential since its left-hand sides can be hierarchically organized as follows:

$$\boxed{n} \leq m \implies \begin{cases} \text{zero} \leq m & \text{(first rule)} \\ \text{succ}(x) \leq \boxed{m} \implies \begin{cases} \text{succ}(x) \leq \text{zero} & \text{(second rule)} \\ \text{succ}(x) \leq \text{succ}(y) & \text{(third rule)} \end{cases} \end{cases}$$

where arguments in a box denote a case distinction (this is similar to the notion of definitional tree in [1]).

The evaluation of terms w.r.t. a TRS is formalized with the notion of *rewriting*. A *rewrite step* is an application of a rewrite rule to a term, i.e.,  $t \rightarrow_{p,R} s$  if there exists a position  $p$  in  $t$ , a rewrite rule  $R = (l \rightarrow r)$  and a substitution  $\sigma$  with  $t|_p = \sigma(l)$  and  $s = t[\sigma(r)]_p$  ( $p$  and  $R$  will often be omitted in the notation of a reduction step). The instantiated left-hand side  $\sigma(l)$  is called a *redex*. A term  $t$  is called *irreducible* or in *normal form* if there is no term  $s$  with  $t \rightarrow s$ . We denote by  $\rightarrow^+$  the transitive closure of  $\rightarrow$  and by  $\rightarrow^*$  its reflexive and transitive closure. Given a TRS  $\mathcal{R}$  and a term  $t$ , we say that  $t$  *evaluates* to  $s$  iff  $t \rightarrow^* s$  and  $s$  is in normal form.

### 3 A Method for Partial Inversion

In this section we present our stepwise method for the partial inversion of inductively sequential TRSs.

#### 3.1 Normalization

The first stage of our transformation is used to *flatten* the right-hand sides of the rules so that no nested function calls occur. This transformation greatly simplifies the definition of the inversion algorithm in Sect. 3.2.

**Definition 1 (normalized TRS).** *A normalized TRS contains rules of the form*

$$l \rightarrow p_0 \quad \text{or} \quad l \rightarrow \mathbf{let} \ p_1 = e_1, \dots, p_n = e_n \ \mathbf{in} \ p_0$$

where  $p_0, p_1, \dots, p_n$  are constructor-terms and  $e_1, \dots, e_n$  are operation-rooted terms with constructor terms as arguments. Each equality,  $p_i = e_i$ , is called a pattern definition. We further require that  $\text{Var}(e_i) \subseteq \text{Var}(l) \cup \text{Var}(p_1) \cup \dots \cup \text{Var}(p_{i-1})$ , for  $i = 1, \dots, n$ , and  $\text{Var}(p_0) \subseteq \text{Var}(l) \cup \text{Var}(p_1) \cup \dots \cup \text{Var}(p_n)$ .<sup>3</sup>

<sup>3</sup> This is similar to the notion of *deterministic* conditional TRS.

In the following, we denote by  $C[e_1, \dots, e_n]$  a term with a constructor context  $C$  and *maximal* operation-rooted subterms  $e_1, \dots, e_n$ . For instance, the term  $c(f(a), s(g(b)))$ , with  $f, g \in \mathcal{D}$  defined functions and  $a, b, c \in \mathcal{C}$  constructor symbols, can be represented by  $C[f(a), g(b)]$ , where the context  $C$  denotes the constructor term  $c(\bullet, s(\bullet))$  with two “holes”. A constructor term (or a variable) can thus be denoted by  $C[\ ]$ , i.e., a term with no maximal operation-rooted subterms.

The following definition introduces the normalization process:

**Definition 2 (normalization).** *Given a TRS  $\mathcal{R}$ , the normalized TRS  $\mathcal{N}(\mathcal{R})$  is obtained by replacing every rewrite rule  $l \rightarrow r \in \mathcal{R}$  by  $l \rightarrow r'$  in  $\mathcal{N}(\mathcal{R})$ , where  $r'$  is obtained from  $r$  by applying the following transformations as much as possible:*

$$\begin{array}{ll}
C[\overline{e_k}] & \Longrightarrow \text{let } x_1 = e_1, \dots, x_k = e_k \text{ in } C[\overline{x_k}] \\
f(\overline{e_k}) & \Longrightarrow \text{let } x = f(\overline{e_k}) \text{ in } x \\
\text{let } p_1 = e_1, & \Longrightarrow \text{let } \overline{x_{jm_j}} = e_{jm_j}, p_1 = e_1, \\
\quad \dots, & \quad \dots, \\
p_i = f(\dots, C[\overline{e_{jm_j}}], \dots) & p_i = f(\dots, C[\overline{x_{jm_j}}], \dots), \\
\quad \dots, & \quad \dots, \\
p_k = e_k \text{ in } p & p_k = e_k \text{ in } p
\end{array}$$

where  $x, x_1, \dots, x_k, x_{j_1}, \dots, x_{j_{m_j}}$  are fresh variables. The process stops when no rule is applicable—clearly a terminating process.

Roughly speaking, normalization proceeds as follows: if the right-hand side is a constructor term then it is already normalized; otherwise,

- If it is an operation-rooted term, then it is completely replaced by a fresh variable and a new pattern definition in a let expression is returned.
- If it is a constructor-rooted term that contains some maximal operation-rooted subterms, normalization replaces those operation-rooted subterms by fresh variables and adds new pattern definitions by means of a let declaration.
- Once the right-hand side is transformed into a let expression, we continue by flattening the arguments of operation-rooted terms in the right-hand sides of pattern definitions so that all function arguments become constructor terms. We note that new pattern definitions are added to the left in order to fulfill the condition on the variables of Def. 1.

Observe that every normalized TRS could be transformed back into an ordinary TRS by applying *inlining*, i.e., by applying the following rules to the right-hand sides of normalized TRSs as much as possible:

$$\begin{array}{l}
\text{let } p_1 = e_1 \text{ in } p \Rightarrow \{p_1 \mapsto e_1\}(p) \\
\text{let } \overline{p_n} = \overline{e_n} \text{ in } p \Rightarrow \{p_1 \mapsto e_1\}(\text{let } p_2 = e_2, \dots, p_n = e_n \text{ in } p) \quad n > 0
\end{array}$$

Note that these rules are well-defined because patterns  $p_i$  are always variables by Def. 2. In general, however, some form of lambda-lifting [7] would be required.

*Example 2.* Consider the following (inductively sequential) TRS that defines the function  $incL$  for incrementing all the elements of a list by a given value:

$$\begin{array}{ll} incL([], i) & \rightarrow [] \\ incL(x : xs, i) & \rightarrow add(i, x) : incL(xs, i) \end{array} \quad \begin{array}{ll} add(zero, y) & \rightarrow y \\ add(succ(x), y) & \rightarrow succ(add(x, y)) \end{array}$$

where lists are built from  $[]$  and  $“:”$ . The normalization of this program returns

$$\begin{array}{ll} incL([], i) & \rightarrow [] \\ incL(x : xs, i) & \rightarrow \mathbf{let} \ w_1 = add(i, x), \\ & \quad \quad \quad w_2 = incL(xs, i) \\ & \quad \quad \quad \mathbf{in} \ w_1 : w_2 \end{array} \quad \begin{array}{ll} add(zero, y) & \rightarrow y \\ add(succ(x), y) & \rightarrow \mathbf{let} \ w = add(x, y) \\ & \quad \quad \quad \mathbf{in} \ succ(w) \end{array}$$

### 3.2 Partial inversion algorithm

The partial inversion of a function  $f$  of arity  $n$  w.r.t. a set  $I = \{i_1, \dots, i_m\} \subset \{1, \dots, n\}$  (the “known” parameters) should return a new function  $\bar{f}_I$  such that  $f(t_1, \dots, t_n)$  evaluates to  $t$  iff  $\bar{f}_I(t, t_{i_1}, \dots, t_{i_m})$  evaluates to  $\langle t_{j_1}, \dots, t_{j_k} \rangle$ , where  $t_1, \dots, t_n, t$  are constructor terms and  $\bar{I} = \{j_1, \dots, j_k\}$  (the “unknown” parameters); here, we denote by  $\bar{I}$  the set  $\{1, \dots, n\} \setminus I$ . Observe that  $I = \{1, \dots, n\}$  is not allowed because it would imply that, in the inverted function, all arguments, together with the output, would be known, which would be meaningless unless one wants to produce a sort “Boolean test”.

#### Algorithm 1 (partial inversion algorithm).

**Input:** a normalized TRS  $\mathcal{R}$ , a function  $f$  of arity  $n$ , and a set  $I \subset \{1, \dots, n\}$ ;

**Output:** a normalized TRS  $\mathcal{R}'$  that includes the definition of the partial inversion of  $f$  w.r.t.  $I$ ;

**Initialization:**  $\mathcal{R}' := \{ \}$ ,  $Inv := \{ \}$ ,  $Pend := \{(f, I)\}$ ;

**Repeat**

1. select a pair  $(f, I) \in Pend$
2. if  $I = \{1, \dots, n\}$  where  $n$  is the arity of  $f$ , then stop with failure; else update  $Inv := Inv \cup \{(f, I)\}$  and  $Pend := Pend \setminus \{(f, I)\}$
3. **if**  $iseq(\mathcal{R}, f, I)$  **then** proceed with step 4 **else** stop with failure
4. let  $\mathcal{R}_f^I = pinv(\mathcal{R}, f, I)$
5. **if**  $\mathcal{R}_f^I = \{ \}$  **then** stop with failure **else**  $\mathcal{R}' := \mathcal{R}' \cup \mathcal{R}_f^I$
6.  $Pend := Pend \cup (pcalls(\mathcal{R}_f^I) \setminus Inv)$

**Until**  $Pend = \{ \}$

**Return**  $\mathcal{R}'$

Roughly speaking, our iterative algorithm for computing a partial inversion of a function proceeds as follows:

- The algorithm takes a normalized program and returns either a failure or a normalized program (the desired partial inversion).
- In every iteration, the partial inversion of a function denoted by a pair  $(f, I)$  is considered, where  $f$  is a function symbol of arity  $n$  and  $I \subset \{1, \dots, n\}$ .

- Then, given a pair  $(f, I)$ , we use the auxiliary Boolean function *iseq* to check whether the new function  $\bar{f}_I$  will fulfill the conditions for inductive sequentiality. Basically, it checks that the left-hand sides in the definition of  $\bar{f}_I$  could be hierarchically organized as in Example 1 (i.e., that a definitional tree [1] exists), that  $\bar{f}_I$  will be left-linear, and that the definition of  $\bar{f}_I$  will not contain extra variables. We note that these conditions can be checked *before* the actual inversion (carried out by function *pinv*) starts.
- Finally, the partial inversion of the considered function is computed by function *pinv*, which returns either a failure (here denoted by the fact that an empty set is returned) or a new definition for  $\bar{f}_I$ . The iteration terminates by updating the set of pending partial inversions with a pair  $(g, J)$  for each call  $\bar{g}_J$  in the definition of  $\bar{f}_I$ ; this is done by using the auxiliary function *pcalls*.

The following definition formalizes the main component of our partial inversion algorithm, function *pinv*:

**Definition 3 (function *pinv*).** *Given a normalized TRS  $\mathcal{R}$ , a function  $f$  of arity  $n$ , and a set  $I \subset \{1, \dots, n\}$ , the partial inversion of  $f$  w.r.t.  $I$ , in symbols  $\text{pinv}(\mathcal{R}, f, I)$ , is obtained as the set*

$$\{ \llbracket l \rightarrow r \rrbracket_I \mid l \rightarrow r \text{ belongs to the definition of } f \text{ in } \mathcal{R} \}$$

if no element in the set contains occurrences of  $\llbracket \ \ \rrbracket_I$  nor  $(( \ \ ))$ , and  $\{ \ \ \}$  otherwise. The auxiliary functions  $\llbracket \ \ \rrbracket_I$  and  $(( \ \ ))$  are defined inductively as follows:

$$\begin{aligned} \llbracket f(\bar{p}_n) \rightarrow C \rrbracket_I &= \bar{f}_I(C[], p_{i_1}, \dots, p_{i_m}) \rightarrow \langle p_{j_1}, \dots, p_{j_k} \rangle \\ \llbracket f(\bar{p}_n) \rightarrow \text{let } \bar{q}_l \equiv e_l \text{ in } C \rrbracket_I &= \bar{f}_I(C[], p_{i_1}, \dots, p_{i_m}) \rightarrow ((\text{let } \bar{q}_l \equiv e_l \\ &\quad \text{in } \langle p_{j_1}, \dots, p_{j_k} \rangle))_V^1 \end{aligned}$$

where  $I = \{i_1, \dots, i_m\}$ ,  $\bar{I} = \{j_1, \dots, j_k\}$ , and  $V = \text{Var}(\bar{f}_I(C[], p_{i_1}, \dots, p_{i_m}))$ .

$$\begin{aligned} ((\text{let } p_1 = e_1, &= ((\text{let } p_1 = e_1, \\ \dots, &\dots, \\ p_l = g(\bar{q}_b) &\langle q_{j_1}, \dots, q_{j_k} \rangle = \bar{g}_I(p_l, q_{i_1}, \dots, q_{i_m}) \\ \dots, &\dots, \\ p_a = e_a \text{ in } p))_V^l &= p_a = e_a \text{ in } p))_{V \cup \text{Var}(q_{j_1}) \cup \dots \cup \text{Var}(q_{j_k})}^{l+1} \\ &\text{if } I = \{i_1, \dots, i_m\} \subset \{1, \dots, b\}, \bar{I} = \{j_1, \dots, j_k\}, \\ &\text{Var}(p_l) \cup \text{Var}(q_{i_1}) \cup \dots \cup \text{Var}(q_{i_m}) \subseteq V, \\ &\text{and } \text{Var}(q_u) \not\subseteq V \text{ for all } u = j_1, \dots, j_k \end{aligned}$$

$$((\text{let } \dots, p_l = e_l, \dots \text{ in } p))_V^l = ((\text{let } \dots, p_l = e_l, \dots \text{ in } p))_{V \cup \text{Var}(p_l)}^{l+1}$$

if  $\text{Var}(e_l) \subseteq V$  and  $p_l \notin V$

$$((\text{let } \bar{p}_a \equiv e_a \text{ in } p))_V^{a+1} = \text{let } \bar{p}_a \equiv e_a \text{ in } p$$

Essentially, function *pinv* above considers sequentially each pattern definition  $p_l = g(q_1, \dots, q_b)$  in the let declaration and transforms it into a new pattern definition according to the set  $V$  of “known” variables (which is initialized to the variables of the left-hand side) as follows:

- if all variables in  $q_1, \dots, q_b$  are known (i.e., belong to  $V$ ) and  $p_l$  (which is a variable by definition of normalized program) is not known (i.e., it does not belong to  $V$ ), then we do not modify this pattern definition;
- otherwise, we divide the parameters of  $g$  into a set  $I$  of known parameters and a set  $\bar{I}$  of unknown parameters, and replace the original pattern definition by  $\langle q_{j_1}, \dots, q_{j_k} \rangle = \bar{g}_I(p_l, q_{i_1}, \dots, q_{i_m})$ , where the known parameters appear as arguments of  $\bar{g}_I$  and the unknown parameters appear as the output of  $\bar{g}_I$ .

*Example 3.* Consider the normalized TRS of Example 2. The stepwise computation of  $\text{pinv}(\mathcal{R}, \text{incL}, \{2\})$  proceeds as follows:

$$\llbracket \text{incL}([], i) \rightarrow [] \rrbracket_{\{2\}} = \overline{\text{incL}}_{\{2\}}([], i) \rightarrow []$$

$$\begin{aligned} & \llbracket \text{incL}(x : xs, i) \rightarrow \mathbf{let} \ w_1 = \text{add}(i, x), \ w_2 = \text{incL}(xs, i) \ \mathbf{in} \ w_1 : w_2 \rrbracket_{\{2\}} \\ & = \overline{\text{incL}}_{\{2\}}(w_1 : w_2, i) \rightarrow ((\mathbf{let} \ w_1 = \text{add}(i, x), \ w_2 = \text{incL}(xs, i) \ \mathbf{in} \ x : xs))_{\{w_1, w_2, i\}}^1 \end{aligned}$$

where

$$\begin{aligned} & ((\mathbf{let} \ w_1 = \text{add}(i, x), \ w_2 = \text{incL}(xs, i) \ \mathbf{in} \ x : xs))_{\{w_1, w_2, i\}}^1 \\ & = ((\mathbf{let} \ x = \overline{\text{add}}_{\{1\}}(w_1, i), \ w_2 = \text{incL}(xs, i) \ \mathbf{in} \ x : xs))_{\{w_1, w_2, i, x\}}^2 \\ & = ((\mathbf{let} \ x = \overline{\text{add}}_{\{1\}}(w_1, i), \ xs = \overline{\text{incL}}_{\{2\}}(w_2, i) \ \mathbf{in} \ x : xs))_{\{w_1, w_2, i, x, xs\}}^3 \\ & = \mathbf{let} \ x = \overline{\text{add}}_{\{1\}}(w_1, i), \ xs = \overline{\text{incL}}_{\{2\}}(w_2, i) \ \mathbf{in} \ x : xs \end{aligned}$$

Now, function  $\text{pcalls}$  would return the set  $\{(\text{add}, \{1\})\}$ . Then, the computation of  $\text{pinv}(\mathcal{R}, \text{add}, \{1\})$  would begin so that the following partial inversion is computed:

$$\begin{aligned} & \overline{\text{add}}_{\{1\}}(y, \text{zero}) \rightarrow y \\ & \overline{\text{add}}_{\{1\}}(\text{succ}(w), \text{succ}(x)) \rightarrow \mathbf{let} \ y = \overline{\text{add}}_{\{1\}}(w, x) \ \mathbf{in} \ y \end{aligned}$$

The final transformed program is thus as follows:

$$\begin{aligned} & \overline{\text{incL}}_{\{2\}}([], i) \rightarrow [] \\ & \overline{\text{incL}}_{\{2\}}(w_1 : w_2, i) \rightarrow \mathbf{let} \ x = \overline{\text{add}}_{\{1\}}(w_1, i), \ xs = \overline{\text{incL}}_{\{2\}}(w_2, i) \ \mathbf{in} \ x : xs \\ & \overline{\text{add}}_{\{1\}}(y, \text{zero}) \rightarrow y \\ & \overline{\text{add}}_{\{1\}}(\text{succ}(w), \text{succ}(x)) \rightarrow \mathbf{let} \ y = \overline{\text{add}}_{\{1\}}(w, x) \ \mathbf{in} \ y \end{aligned}$$

which implements a function  $\overline{\text{incL}}_{\{2\}}$  that decrements all the elements of the input list by a given value (using the auxiliary function  $\overline{\text{add}}_{\{1\}}$  to perform subtraction on natural numbers).

### 3.3 Removal of let declarations

Let expressions in transformed programs can easily be removed by applying a simplified version of *lambda lifting* [7]. In particular, we follow the transformation presented in [4, Appendix D], where a rule of the form

$$l \rightarrow \mathbf{let} \ p_1 = e_1, \dots, p_{i-1} = e_{i-1}, p_i = e_i, p_{i+1} = e_{i+1}, \dots, p_m = e_m \ \mathbf{in} \ e$$

is transformed into the rules

$$\begin{array}{lcl}
l & \rightarrow & g(\overline{x_k}, e_i) \\
g(\overline{x_k}, z) & \rightarrow & g'(\overline{x_k}, g_1(z), \dots, g_m(z)) \\
g'(\overline{x_k}, \overline{y_m}) & \rightarrow & \mathbf{let} \ p_1 = e_1, \dots, p_{i-1} = e_{i-1}, p_{i+1} = e_{i+1}, \dots, p_m = e_m \ \mathbf{in} \ e \\
g_1(p_i) & \rightarrow & y_1 \\
\dots & & \\
g_m(p_i) & \rightarrow & y_m
\end{array}$$

where  $x_1, \dots, x_k$  are the variables of  $l$ ,  $y_1, \dots, y_m$  are the variables occurring in  $p_i$ ,  $z$  is a fresh variable, and  $g, g', g_1, \dots, g_m$  are new function symbols. This step is repeated until all local patterns are eliminated.

Nevertheless, we allow the application of the simpler transformation of *inlining* when the pattern definition has the form  $x = e$ .

*Example 4.* Consider the partially inverted TRS of Example 3. Here, inlining suffices to remove let expressions, so that the following standard TRS is obtained:

$$\begin{array}{lcl}
\overline{incL}_{\{2\}}([], i) & \rightarrow & [] \\
\overline{incL}_{\{2\}}(w_1 : w_2, i) & \rightarrow & \overline{add}_{\{1\}}(w_1, i) : \overline{incL}_{\{2\}}(w_2, i) \\
\overline{add}_{\{1\}}(y, zero) & \rightarrow & y \\
\overline{add}_{\{1\}}(succ(w), succ(x)) & \rightarrow & \overline{add}_{\{1\}}(w, x)
\end{array}$$

## 4 Related Work

Among the closest approaches, we have the work by Glück and Kawabe [5] (further improved in [6]), where an automatic program inversion algorithm for first-order functional programs is presented. In contrast to ours, a *total* inversion algorithm is considered (a particular case of our partial inversion method) and, thus, only *injective* functions are allowed.

The closest approach is that of Nishida et al. [9], where the authors present a very general inversion algorithm for term rewriting systems which is able to perform both partial and total inversions. The main differences with our approach are as follows:

- The method of [9] only preserve correctness w.r.t. eager computations. For instance, given the following (infinite) function:

$$from(x) \rightarrow x : from(succ(x))$$

they would return the (total) inversion  $\overline{from}_{\emptyset}$ :

$$\begin{array}{lcl}
\overline{from}_{\emptyset}(x : y) & \rightarrow & \mathbf{let} \ \langle \rangle = \overline{from}_{\{1\}}(y, succ(x)) \ \mathbf{in} \ x \\
\overline{from}_{\{1\}}(x : y, x) & \rightarrow & \mathbf{let} \ \langle \rangle = \overline{from}_{\{1\}}(y, succ(x)) \ \mathbf{in} \ \langle \rangle
\end{array}$$

Here, the pattern definition  $\langle \rangle = \overline{from}_{\{1\}}(y, succ(x))$  is only used to test that  $y$  is a list headed by  $succ(x)$ . However, under a lazy semantics, this

condition would never be evaluated and, thus, we have that  $\overline{from}_{\emptyset}(zero : zero)$  evaluates to  $zero$  while  $from(zero)$  does not evaluate to  $zero : zero$ .

This situation would not happen in our method because the inversion of function  $\overline{from}_{\emptyset}$  requires the computation of  $\overline{from}_{\{1\}}$ , which is not allowed since  $\{1\}$  contains all the parameters of  $from$ . Therefore, we would return a failure.

- The (more general) inversion technique of [9] introduces some additional rules that are not needed in our approach. For instance, in order to preserve the correctness, [9] will add the following rule:

$$\overline{add}_{\{2\}}(add(x, y), y) \rightarrow \langle x \rangle$$

to the definition of  $\overline{add}_{\{2\}}$ .

Furthermore, they require a form of *narrowing* [10] to perform computations in the inverted program, while functional reduction suffices in our case.

- Another difference comes from the way in which each pattern definition is inverted. In [9], the lists of known and unknown variables is fixed at the beginning of a rule (to be processed) and is not changed until the partial inversion of the rule ends. In our case, this set is dynamically augmented during the process. For instance, the following function:

$$g(x) \rightarrow mul(x, half(succ(x)))$$

is inverted in [9] as follows:

$$\overline{g}_{\emptyset}(y) \rightarrow \mathbf{let} \langle x, z \rangle = \overline{mul}_{\emptyset}(y), \langle succ(x) \rangle = \overline{half}_{\emptyset}(z) \mathbf{in} \langle x \rangle$$

Here, the fact that variable  $x$  is known after the evaluation of  $\langle x, z \rangle = \overline{mul}_{\emptyset}(y)$  is not taken into account when inverting the last pattern definition. In contrast, our method would return

$$\overline{g}_{\emptyset}(y) \rightarrow \mathbf{let} \langle x, z \rangle = \overline{mul}_{\emptyset}(y), \langle \rangle = \overline{half}_{\{1\}}(z, succ(x)) \mathbf{in} \langle x \rangle$$

which would imply a failure because  $\{1\}$  are all the parameters of function  $half$ .

Moreover, in our method, some functions could remain unchanged when their inversion is not needed—i.e., the second case of function  $(( \ ))$ —while this is not allowed in [9].

To summarize, our method is simpler than that of [9] and can be applied in fewer cases, but when it succeeds, the resulting program is inductively sequential and preserves computations under a lazy evaluation.

Finally, Mogensen [8] has recently introduced a method for computing the *semi-inversion* of a functional program with guarded equations. Basically, semi-inversion means taking a program and producing a new program that as input takes *part* of the input and *part* of the output of the original program and as output produces the rest of the input and output of the original program. This work tackles a more general objective than ours, but the underlying techniques are also similar to both ours and that of [9].

## 5 Discussion

We have presented a novel method for the partial inversion of inductively sequential rewrite systems. When the method succeeds, it returns an inductively sequential system without extra variables, which is essential to have a practically applicable method.

We are currently working on the formal proof of correctness for the transformation (by extending the results in [9]) as well as on the implementation of the partial inversion technique.

## References

1. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.
2. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
3. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
4. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: <http://www.informatik.uni-kiel.de/~mh/curry/>.
5. R. Glück and M. Kawabe. A Program Inverter for a Functional Language with Equality and Constructors. In *Proc. of the 1st Asian Symp. on Programming Languages and Systems (APLAS'03)*, pages 246–264. Springer LNCS 2895, 2003.
6. R. Glück and M. Kawabe. Derivation of deterministic inverse programs based on LR parsing. In *Proc. of the 7th Int'l Symp. on Functional and Logic Programming (FLOPS'04)*, pages 291–306. Springer LNCS 2998, 2004.
7. T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Functional Programming Languages and Computer Architecture (Nancy, France)*, pages 190–203. Springer LNCS 201, 1985.
8. T.Æ. Mogensen. Semi-inversion of Guarded Equations. In *In Proc. of the 4th Int'l Conf. on Generative Programming and Component Engineering (GPCE'05)*, pages 189–204. Springer LNCS 3676, 2005.
9. N. Nishida, M. Sakai, and T. Sakabe. Partial Inversion of Constructor Term Rewriting Systems. In *Proc. of the 16th Int'l Conf. on Term Rewriting and Applications (RTA 2005)*, pages 264–278. Springer LNCS 3467, 2005.
10. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.