Automatic Partial Inversion of Inductively Sequential Functions[⋆]

Jesús M. Almendros-Jiménez[1] and Germán Vidal[2]

[1] University of Almería, Spain. Email: jalmen@ual.es
[2] 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). To ease the definition, our method proceeds in a stepwise manner: normalization (introduction of let expressions), proper inversion, and removal of let expressions. Furthermore, it can easily be implemented. Therefore, it forms an appropriate basis for developing a practically applicable transformation tool. Preliminary experiments with a prototype implementation of the partial inverter demonstrates the usefulness and viability of our approach.

1 Introduction

Program inversion is a fundamental transformation within the functional programming paradigm. Having a fully automatic inversion tool could be very useful for programmers because there are many functions that can be seen as the inverse of other, sometimes easier, functions (e.g., encoding and decoding, compression and decompression, etc). Moreover, having a function and its inverse can also be useful for defining *views* [19], where one needs to implement translation functions from a built-in data type to an algebraic data type and vice versa, so that both functions are inverses of each other.

Intuitively speaking, given a function f of arity n, the *total inversion* of function f is a new function f^{-1} such that

$$f^{-1}(t) = \langle t_1, \ldots, t_n \rangle \qquad \text{if and only if} \qquad f(t_1, \ldots, t_n) = t$$

for all terms $t_1, \ldots, t_n, t$. Computing the total inversion of a function is a difficult task and, in most cases, the inverse of a function does not exist (e.g., when the given function is not injective).

In this paper, and in contrast to most of the previous work on program inversion, we consider the computation of *partial inverses.* Roughly speaking,

[⋆] 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.

given a function f, the *partial inversion* of f w.r.t. the set of parameters $I = \{i_1, \ldots, i_m\} \subset \{1, \ldots, n\}$ is a new function $\overline{f}_I$ such that

$$\overline{f}_I(t, t_{i_1}, \ldots, t_{i_m}) = \langle t_{j_1}, \ldots, t_{j_k} \rangle \quad \text{if and only if} \quad f(t_1, \ldots, t_n) = t$$

for all terms $t_1, \ldots, t_n, t$, with $\{j_1, \ldots, j_k\} = \{1, \ldots, n\} \setminus I$. Clearly, partial inversion subsumes total inversion (when $I = \varnothing$). In contrast to total inversion, however, the considered function needs not be injective in order to be acceptable for partial inversion. Nevertheless, some form of injectivity w.r.t. the parameters I is required (see Sect. 3.1).

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

$$\begin{array}{lll} \mathit{add}(\mathit{zero}, y) & \to & y \\ \mathit{add}(\mathit{succ}(x), y) & \to & \mathit{succ}(\mathit{add}(x, y)) \end{array}$$

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

$$\begin{array}{rll} \overline{\mathit{add}}_\varnothing(t) = \langle t_1, t_2 \rangle & \Leftrightarrow & \mathit{add}(t_1, t_2) = t \\ \overline{\mathit{add}}_{\{1\}}(t, t_1) = t_2 & \Leftrightarrow & \mathit{add}(t_1, t_2) = t \\ \overline{\mathit{add}}_{\{2\}}(t, t_2) = t_1 & \Leftrightarrow & \mathit{add}(t_1, t_2) = t \end{array}$$

Their definitions can be given, respectively, as follows:

$$\begin{array}{lll} \overline{\mathit{add}}_\varnothing(y) & \to & \langle \mathit{zero}, y \rangle \\ \overline{\mathit{add}}_\varnothing(\mathit{succ}(w)) & \to & \mathbf{let}\ \langle x, y \rangle = \overline{\mathit{add}}_\varnothing(w)\ \mathbf{in}\ \langle \mathit{succ}(x), y \rangle \\ & & \\ \overline{\mathit{add}}_{\{1\}}(y, \mathit{zero}) & \to & y \\ \overline{\mathit{add}}_{\{1\}}(\mathit{succ}(w), \mathit{succ}(x)) & \to & \overline{\mathit{add}}_{\{1\}}(w, x) \\ & & \\ \overline{\mathit{add}}_{\{2\}}(y, y) & \to & \mathit{zero} \\ \overline{\mathit{add}}_{\{2\}}(\mathit{succ}(w), y) & \to & \mathit{succ}(\overline{\mathit{add}}_{\{2\}}(w, y)) \end{array}$$

Observe that both $\overline{\mathit{add}}_{\{1\}}$ and $\overline{\mathit{add}}_{\{2\}}$ define the subtraction on natural numbers (though they are syntactically different).

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 does not have overlapping left-hand sides (i.e., no left-hand sides unify). However, in the above partial inversions,

– the definition of the partial inverse $\overline{\mathit{add}}_\varnothing$ has overlapping left-hand sides, and
– the definition of the partial inverse $\overline{\mathit{add}}_{\{2\}}$ 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, we consider that ensuring that partially inverted programs are inductively sequential (as the original ones) is mandatory, since otherwise the practical applicability of these partially inverted functions is unclear. For instance, although $\overline{add}_{\{1\}}$ and $\overline{add}_{\{2\}}$ are semantically equivalent (in the sense that both implement subtraction: $\overline{add}_{\{1\}}(t, t_1) = t_2$ iff $\overline{add}_{\{2\}}(t, t_2) = t_1$), the first function $\overline{add}_{\{1\}}$ can be used in any functional programming language or environment, while the second one $\overline{add}_{\{2\}}$ is often illegal (e.g., in Haskell) because it is not left-linear.

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., while function $\overline{add}_{\{1\}}$ implements the subtraction on natural numbers, the practical use of the total inverse $\overline{add}_{\varnothing}$ 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. As a consequence, it can be efficiently implemented.
– Finally, our method always terminates, either returning an inductively sequential program—defining the partial inversion of a function—or a failure.

2 Preliminaries

We follow the standard framework of *term rewriting* [2] since it suffices to model the first-order component of many functional programming languages.

Term Rewriting Systems. In term rewriting, a set of rewrite rules (or oriented equations) $l \to r$ 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 $\mathcal{C} = \mathcal{F} \backslash \mathcal{D}$. We often write f/n to denote that the arity of the function or constructor f is n. 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 $\mathcal{F} \cap \mathcal{V} = \varnothing$.

A TRS $\mathcal{R}$ is *constructor-based* if the left-hand sides of its rules have the form $f(s_1, \ldots, s_n)$ where s_i are constructor terms, i.e., $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, for all $i = 1, \ldots, n$. The set of variables appearing in a term t is denoted by $\mathcal{V}ar(t)$. 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 $l \to r \in \mathcal{R}$. 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 $root(t)$. A term t is *operation-rooted* (resp. *constructor-rooted*) if $root(t) \in \mathcal{D}$ (resp. $root(t) \in \mathcal{C}$). As it is common practice, a *position* p in a term t is represented by a sequence of natural numbers, where ϵ 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 $\mathcal{V}ar(t) = \varnothing$. A *substitution* σ is a mapping $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ from variables to terms such that its domain $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is finite. The identity substitution is denoted by *id*. We write $\overline{o_n}$ for the *sequence of syntactic objects* $o_1, \ldots, o_n$.

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 σ 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). 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.

Inductively Sequential Systems. Inductively sequential TRSs [1] are a subclass of constructor-based left-linear TRSs. The formal definition of this class of programs requires the notion of *definitional tree* [1]. Essentially, a TRS is *inductively sequential* [1] 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 (i.e., a typical functional program).

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

$$\begin{array}{rcll} zero & \leqslant & y & \rightarrow & true \\ succ(x) & \leqslant & zero & \rightarrow & false \\ succ(x) & \leqslant & succ(y) & \rightarrow & x \leqslant y \end{array}$$

This function is inductively sequential because the left-hand sides can be inductively organized as follows:[3]

$$\boxed{n} \leqslant m \Rightarrow \begin{cases} zero \leqslant m \;\rightarrow\; true & \text{(first rule)} \\ succ(x) \leqslant \boxed{\text{m}} \Rightarrow \begin{cases} succ(x) \leqslant zero \;\rightarrow\; false & \text{(second rule)} \\ succ(x) \leqslant succ(y) \;\rightarrow\; x \leqslant y & \text{(third rule)} \end{cases} \end{cases}$$

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.

[3] Actually, this is the *definitional tree* of function "$\leqslant$".

3 A Method for Partial Inversion

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

In the following, we consider the partial inversion of a given function f/n w.r.t. a set $I \subset \{1,\ldots,n\}$ of input (or "known") parameters. Therefore, we want to obtain a new function, which we call $\overline{f}_I$, which takes the output of the original function and the input parameters (according to I), and returns the remaining parameters of the original function, which we denote by $\bar{I} = \{1,\ldots,n\}\backslash I$ (the "unknown" parameters).

Observe that $I = \{1,\ldots,n\}$ 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 of "Boolean test". Now, we formally introduce our notion of partial inversion:

Definition 1 (partial inversion). *Let $\mathcal{R}$ be an inductively sequential TRS that includes the definition of function f/n. Then, $\mathcal{R}'$ is a partial inversion of $\mathcal{R}$ w.r.t. f and $I = \{i_1,\ldots,i_m\} \subset \{1,\ldots,n\}$ iff the following conditions hold:*

1. *$\mathcal{R}'$ is inductively sequential and*
2. *it includes the definition of a function $\overline{f}_I$ such that $f(t_1,\ldots,t_n) \to^* t$ iff $\overline{f}_I(t,t_{i_1},\ldots,t_{i_m}) \to^* \langle t_{j_1},\ldots,t_{j_k}\rangle$ for all ground constructor terms $t_1,\ldots,t_n$, t, where $\bar{I} = \{j_1,\ldots,j_k\}$.*

In this case, we say that $\overline{f}_I$ is the partial inverse of f w.r.t. I.

As mentioned before, the first condition above is often ignored (e.g., [15]), but we require it in order to produce partially inverted programs which are useful in practice.

3.1 Preconditions

In this section, we present three preconditions for our partial inversion algorithm to be successful. These preconditions are *local*, i.e., should be checked for every function involved in the partial inversion process (see Sect. 3.3).

As mentioned in the introduction, functions need not be injective to be partially inverted. However, some form of injectivity is still necessary. Let us consider a function f/n that we want to partially invert w.r.t. $I = \{i_1,\ldots,i_m\} \subset \{1,\ldots,n\}$. Assume a relation $Rel(f)$, defined as follows:

$$Rel(f) = \{(t_1,\ldots,t_n,t) \mid f(t_1,\ldots,t_n) \to^* t\}$$

where $t_1,\ldots,t_n,t$ are ground constructor terms (i.e., *values*). Then, we say that the partial inversion of f w.r.t. I is *well-defined* if $(t_{j_1},\ldots,t_{j_k}) \neq (s_{j_1},\ldots,s_{j_k})$ implies $(t_{i_1},\ldots,t_{i_m},t) \neq (s_{i_1},\ldots,s_{i_m},s)$ for all tuples $(t_1,\ldots,t_n,t), (s_1,\ldots,s_n,t)$ in $Rel(f)$, where $\bar{I} = \{j_1,\ldots,j_k\}$.

Trivially, a total inversion is well-defined when the considered function is injective. In general, however, the set of functions that can be partially inverted is greater than the set of functions that can be totally inverted.

For instance, the total inversion of the addition function *add* shown in Sect. 1 is not well-defined because *add* is not injective. On the other hand, the partial inversion of *add* w.r.t. $\{1\}$ is well-defined because, given the evaluations $add(t_1, t_2) \to^* t$ and $add(s_1, s_2) \to^* s$, whenever $t_2 \neq s_2$, we have $(t_1, t) \neq (s_1, s)$, where t_1, t_2, t, s_1, s_2, s are ground constructor terms.

Unfortunately, determining if a partial inversion is well-defined is generally undecidable. Therefore, we introduce three (decidable) preconditions for partial inversion. The first precondition, which regards extra variables, is very simple:

Precondition 1 (extra variables). Let f/n be a function to be partially inverted w.r.t. $I = \{i_1, \ldots, i_m\} \subset \{1, \ldots, n\}$. Then, function f/n must fulfill the following condition: $\mathcal{V}ar(\{t_{j_1}, \ldots, t_{j_k}\}) \subseteq \mathcal{V}ar(\{r, t_{i_1}, \ldots, t_{i_m}\})$ for every rule $f(t_1, \ldots, t_n) \to r$ in the definition of f, with $\bar{I} = \{j_1, \ldots, j_k\}$.

For instance, a function *fst* defined by a rule of the form $fst(x, y) \to x$ cannot be partially inverted w.r.t. $\{1\}$ since $\mathcal{V}ar(\{y\}) \not\subseteq \mathcal{V}ar(\{x, x\})$. Indeed, the definition of the partially inverted function $\overline{fst}_{\{1\}}$ would contain an extra variable: $\overline{fst}_{\{1\}}(x, x) \to y$.

In the following, we denote by $C[e_1, \ldots, e_n]$ a term with a constructor context C and *maximal* operation-rooted subterms $e_1, \ldots, e_n$. 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 second precondition regards left-linearity and is also rather simple:

Precondition 2 (left-linearity). Let f/n be a function to be partially inverted w.r.t. $I = \{i_1, \ldots, i_m\} \subset \{1, \ldots, n\}$. Then, C must be linear and must not share variables with $t_{i_1}, \ldots, t_{i_m}$ for every rule $f(t_1, \ldots, t_n) \to C[e_1, \ldots, e_l]$ in the definition of function f.

Consider, e.g., the following function *double*:

$$\begin{aligned} double([\,]) &\to [\,] \\ double(x : xs) &\to x : x : double(xs) \end{aligned}$$

where lists are built from $[\,]$ and ":". This function does not fulfill the second precondition because the constructor part in the right-hand side of the second rule, $x : x : \bullet$, is not linear. Actually, the partial inversion of *double* w.r.t. $\varnothing$ would return the following rules:

$$\begin{aligned} \overline{double}_\varnothing([\,]) &\to [\,] \\ \overline{double}_\varnothing(x : x : xs) &\to x : \overline{double}_\varnothing(xs) \end{aligned}$$

Also, the partial inversion of function *fst* w.r.t. $\{1\}$ above does not fulfill the second precondition because the right-hand side x is linear but also occur in the first input parameter x. On the other hand, the second precondition holds for function *fst* w.r.t. $\{2\}$ since x and y do not share variables.

We note that the second precondition could be removed by allowing the replacement of repeated occurrences of the same variable in the left-hand side of a rule by equality tests in the corresponding right-hand side. For example, the definition of $\overline{double}_\varnothing$ could be transformed as follows:

$$\begin{array}{r}\overline{double}_\varnothing([\,]) \to [\,] \\ \overline{double}_\varnothing(x : y : xs) \to cond(eq(x,y), x : \overline{double}_\varnothing(xs))\end{array}$$

where $cond(c,t)$ returns t if c evaluates to *true* and $eq(t_1,t_2)$ is a Boolean equality test. Such a transformation, however, would not be useful in a lazy context because *eq* should be regarded as a *strict* equality and, thus, the inverted function would be more strict than the original function. It could be useful in the context of a strict language though.

We now present our last precondition for ensuring the inductive sequentiality of the partially inverted function.

Precondition 3 (inductive sequentiality). Let f/n be a function to be partially inverted w.r.t. $I = \{i_1,\ldots,i_m\} \subset \{1,\ldots,n\}$. Then, there must be a definitional tree for a function $\overline{f}_I$ whose definition contains the following left-hand sides:

$$\{\ \overline{f}_I(C[x_1,\ldots,x_l], t_{i_1},\ldots,t_{i_m}) \mid \begin{array}{l} f(t_1,\ldots,t_n) \to C[e_1,\ldots,e_l] \in \mathcal{R}_f \\ \text{and } x_1,\ldots,x_l \text{ are fresh variables} \end{array}\ \}$$

where $\mathcal{R}_f$ contains the rules in the definition of function f.

Observe that the above precondition can be tested *before* partial inversion proceeds, since only the left-hand sides are relevant to determine the existence of a definitional tree associated to a function.

Consider, for instance, the following function *app*:

$$\begin{array}{r}app([\,],y) \to y \\ app(x : xs, y) \to x : app(xs,y)\end{array}$$

If we consider its partial inversion w.r.t. $\{2\}$, then the third precondition does not hold since there is no definitional tree for a function defined by a set of rules whose left-hand sides are $\{\overline{app}_{\{2\}}(y,y),\ \overline{app}_{\{2\}}(x : w, y)\}$ (roughly speaking, because the left-hand sides overlap).

Now, we present our stepwise process for partial inversion.

3.2 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 is not really necessary for partially inverting functions, but it greatly simplifies the definition of the inversion algorithm in Sect. 3.3.

Definition 2 (normalized TRS). *A normalized TRS contains either rules of the form*

$$l \;\rightarrow\; p_0 \qquad or \qquad l \;\rightarrow\; \mathbf{let}\; p_1 = e_1, \ldots, p_n = e_n \;\mathbf{in}\; p_0$$

where $p_0, p_1, \ldots, p_n$ are constructor terms and $e_1, \ldots, e_n$ are operation-rooted terms with constructor terms as arguments (i.e., nested defined function symbols are not allowed). Each equality, $p_i = e_i$, is called a pattern definition*. We further require that $\mathcal{V}ar(e_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \ldots \cup \mathcal{V}ar(p_{i-1})$, for $i = 1, \ldots, n$, and $\mathcal{V}ar(p_0) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \ldots \cup \mathcal{V}ar(p_n)$.*[4]

Although let expressions may introduce extra variables, these are a kind of *local* variables that can easily be removed by either inlining or lambda lifting (see below). The following definition introduces our normalization process:

Definition 3 (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:*

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

where $x, x_1, \ldots, x_k, x_{j1}, \ldots, x_{jm_j}$ 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. 2.

[4] This is similar to the notion of *deterministic* conditional TRS.

```
**Input:** a normalized TRS $\mathcal{R}$, a function $f/n$, and a set $I \subset \{1,\ldots,n\}$;
**Output:** a normalized TRS $\mathcal{R}'$ (the partial inversion of $\mathcal{R}$ w.r.t. $f$ and $I$) or a failure;
**Initialization:** $\mathcal{R}' := \{\ \},\ \mathit{Inv} := \{\ \},\ \mathit{Pend} := \{(f/n, I)\}$;
**Repeat**
    1. select a pair $(f/n, I) \in \mathit{Pend}$
    2. **if** $I = \{1,\ldots,n\}$
       **then** stop with failure;   /* Boolean tests are not allowed */
       **else** update $\mathit{Inv} := \mathit{Inv} \cup \{(f,I)\}$ and $\mathit{Pend} := \mathit{Pend} \setminus \{(f,I)\}$
    3. **if** the Preconditions 1, 2 and 3 hold
       **then** proceed with step 4
       **else** stop with failure   /* $\mathcal{R}'$ would not be inductively sequential */
    4. let $\mathcal{R}^I_f = \mathit{pinv}(\mathcal{R}, f, I)$; update $\mathcal{R}' := \mathcal{R}' \cup \mathcal{R}^I_f$
    5. $\mathit{Pend} := \mathit{Pend} \cup (\mathit{pcalls}(\mathcal{R}^I_f) \setminus \mathit{Inv})$
**Until** $\mathit{Pend} = \{\ \}$
**Return** $\mathcal{R}'$
```

Fig. 1. Partial inversion algorithm

Observe that, if we take a TRS and normalize it using Def. 3, then it 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:

$$\begin{array}{lll} \textbf{let}\ p_1 = e_1\ \textbf{in}\ p & \Rightarrow & \{p_1 \mapsto e_1\}(p) \\ \textbf{let}\ \overline{p_n = e_n}\ \textbf{in}\ p & \Rightarrow & \{p_n \mapsto e_n\}(\textbf{let}\ p_1 = e_1, \ldots, p_{n-1} = e_{n-1}\ \textbf{in}\ p) \quad n > 1 \end{array}$$

Note that these rules are well-defined in our case because patterns p_i are always variables in TRSs obtained by applying Def. 3. In general, however, some form of lambda-lifting [9] is required to remove let expressions (see Sect. 3.4).

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:

$$\begin{array}{lll@{\qquad}lll} \mathit{incL}([\,], i) & \to & [\,] & \mathit{add}(\mathit{zero}, y) & \to & y \\ \mathit{incL}(x : xs, i) & \to & \mathit{add}(i, x) : \mathit{incL}(xs, i) & \mathit{add}(\mathit{succ}(x), y) & \to & \mathit{succ}(\mathit{add}(x, y)) \end{array}$$

The normalization of this program returns

$$\begin{array}{lll@{\qquad}lll} \mathit{incL}([\,], i) & \to & [\,] & \mathit{add}(\mathit{zero}, y) & \to & y \\ \mathit{incL}(x : xs, i) & \to & \textbf{let}\ w_1 = \mathit{add}(i, x), & \mathit{add}(\mathit{succ}(x), y) & \to & \textbf{let}\ w = \mathit{add}(x, y) \\ & & \quad\ \ w_2 = \mathit{incL}(xs, i) & & & \textbf{in}\ \mathit{succ}(w) \\ & & \textbf{in}\ w_1 : w_2 & & & \end{array}$$

3.3 Partial inversion algorithm

Our algorithm for partial inversion is shown in Fig. 1. Roughly speaking, our iterative algorithm for computing the partial inversion of a function proceeds as follows:

$$
\begin{array}{ll}
(\!(\mathbf{let}\ \ldots, p_l = e_l, \ldots\ \mathbf{in}\ p)\!)^l_V & = (\!(\mathbf{let}\ \ldots, p_l = e_l, \ldots\ \mathbf{in}\ p)\!)^{l-1}_{V \cup \mathcal{V}ar(p_l)} \\
 & \quad \text{if } \mathcal{V}ar(e_l) \subseteq V \text{ and } p_l \not\in V \\
(\!(\mathbf{let}\ p_1 = e_1, & = (\!(\mathbf{let}\ p_1 = e_1, \\
\qquad \ldots, & \qquad \ldots, \\
\qquad p_l = g(\overline{q_b}) & \qquad \langle q_{j_1}, \ldots, q_{j_k} \rangle = \overline{g}_{\{i_1,\ldots,i_m\}}(p_l, q_{i_1}, \ldots, q_{i_m}) \\
\qquad \ldots, & \qquad \ldots, \\
\qquad p_a = e_a\ \mathbf{in}\ p)\!)^l_V & \qquad p_a = e_a\ \mathbf{in}\ p)\!)^{l-1}_{V \cup \mathcal{V}ar(q_{j_1}) \cup \ldots \cup \mathcal{V}ar(q_{j_k})} \\
 & \quad \text{if } \mathcal{V}ar(q_w) \subseteq V \text{ for all } w = i_1, \ldots, i_m,\ m \geq 0, \\
 & \quad\ \ \mathcal{V}ar(q_u) \not\subseteq V \text{ for all } u = j_1, \ldots, j_k,\ k \geq 0, \text{and} \\
 & \quad\ \ \{i_1, \ldots, i_m\} \uplus \{j_1, \ldots, j_k\} = \{1, \ldots, b\} \\
(\!(\mathbf{let}\ \overline{p_a = e_a}\ \mathbf{in}\ p)\!)^0_V & = \mathbf{let}\ \overline{p_a = e_a}\ \mathbf{in}\ p
\end{array}
$$

Fig. 2. Auxiliary function $(\!(\ )\!)$

- 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 $(f/n, I)$ is considered, where f is a function symbol of arity n and $I \subset \{1, \ldots, n\}$.
- Given such a pair $(f/n, I)$, we first check the preconditions of Sect. 3.1 in order to stop the inversion process if the partial inversion of f w.r.t. I would not be inductively sequential (with no extra variables).
- If the preconditions hold, then we compute the partial inversion $\overline{f}_I$ of f w.r.t. I by means of function *pinv* (see Def. 4).
- The iteration terminates by updating the set of pending partial inversions; this is done by using the auxiliary function *pcalls*, which simply traverses the right-hand sides of a function definition and then returns a set which includes a pair $(g/m, J)$ for each call $\overline{g}_J(t_1, \ldots, t_m)$ in these right-hand sides.

The following definition formalizes the main component of our partial inversion algorithm:

Definition 4 (function *pinv***).** *Let $\mathcal{R}$ be a normalized TRS, f/n be a function, and $I \subset \{1, \ldots, n\}$ be a set. The partial inversion of f w.r.t. I, in symbols $pinv(\mathcal{R}, f, I)$, is obtained as the set*

$$\{\ [\![l \to r]\!]_I \mid l \to r \text{ belongs to the definition of } f \text{ in } \mathcal{R}\ \}$$

Function $[\![\ ]\!]$ is defined as follows:

$$
\begin{array}{ll}
[\![f(\overline{p_n}) \to C[]\,]\!]_I & = \overline{f}_I(C[], p_{i_1}, \ldots, p_{i_m}) \to \langle p_{j_1}, \ldots, p_{j_k} \rangle \\
[\![f(\overline{p_n}) \to \mathbf{let}\ \overline{q_l = e_l}\ \mathbf{in}\ C[]\,]\!]_I & = \overline{f}_I(C[], p_{i_1}, \ldots, p_{i_m}) \to (\!(\mathbf{let}\ \overline{q_l = e_l} \\
 & \qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \langle p_{j_1}, \ldots, p_{j_k} \rangle)\!)^l_V
\end{array}
$$

where $I = \{i_1, \ldots, i_m\}$, $\bar{I} = \{j_1, \ldots, j_k\}$, and $V = \mathcal{V}ar(\overline{f}_I(C[], p_{i_1}, \ldots, p_{i_m}))$. The auxiliary function $(\!(\ )\!)$ is defined inductively as shown in Fig. 2.

Essentially, function *pinv* above considers sequentially[5] each pattern definition $p_l = g(q_1, \ldots, q_b)$ 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 new left-hand side) as follows:

– If all variables in $q_1, \ldots, q_b$ are known (i.e., belong to V), then we do not modify this pattern definition (i.e., a call to a function of the original program is performed);
– Otherwise, we divide the parameters of g into a set $\{i_1, \ldots, i_m\}$ of input parameters—i.e., associated to those arguments of g whose variables belong to the current set V of "known" variables—and output parameters $\{j_1, \ldots, j_k\}$, and replace the original pattern definition by $\langle q_{j_1}, \ldots, q_{j_k}\rangle = \overline{g}_{\{i_1,\ldots,i_m\}}(p_l, q_{i_1}, \ldots, q_{i_m})$.

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

$$[\![\mathit{incL}([\,], i) \to [\,]]\!]_{\{2\}} \quad = \quad \overline{\mathit{incL}}_{\{2\}}([\,], i) \to [\,]$$

$$\begin{array}{l}[\![\mathit{incL}(x : xs, i) \to \mathbf{let}\ w_1 = \mathit{add}(i, x),\ w_2 = \mathit{incL}(xs, i)\ \mathbf{in}\ w_1 : w_2]\!]_{\{2\}} \\ =\ \overline{\mathit{incL}}_{\{2\}}(w_1 : w_2, i) \to (\!(\mathbf{let}\ w_1 = \mathit{add}(i, x),\ w_2 = \mathit{incL}(xs, i)\ \mathbf{in}\ x : xs)\!)^2_{\{w_1,w_2,i\}}\end{array}$$

where

$$\begin{array}{l}(\!(\mathbf{let}\ w_1 = \mathit{add}(i, x),\ w_2 = \mathit{incL}(xs, i)\ \mathbf{in}\ x : xs)\!)^2_{\{w_1,w_2,i\}} \\ \quad =\ (\!(\mathbf{let}\ w_1 = \mathit{add}(i, x),\ xs = \overline{\mathit{incL}}_{\{2\}}(w_2, i)\ \mathbf{in}\ x : xs)\!)^1_{\{w_1,w_2,i,xs\}} \\ \quad =\ (\!(\mathbf{let}\ x = \overline{\mathit{add}}_{\{1\}}(w_1, i),\ xs = \overline{\mathit{incL}}_{\{2\}}(w_2, i)\ \mathbf{in}\ x : xs)\!)^0_{\{w_1,w_2,i,xs,x\}} \\ \quad =\ \mathbf{let}\ x = \overline{\mathit{add}}_{\{1\}}(w_1, i),\ xs = \overline{\mathit{incL}}_{\{2\}}(w_2, i)\ \mathbf{in}\ x : xs\end{array}$$

Now, function *pcalls* would return the set $\{(\mathit{add}/2, \{1\}),\ (\mathit{incL}/2, \{2\})\}$, though only $(\mathit{add}/2, \{1\})$ is added to *Pend* since $(\mathit{incL}/2, \{2\})$ already belongs to *Inv*. Then, the computation of $\mathit{pinv}(\mathcal{R}, \mathit{add}, \{1\})$ begins so that the following partial inversion is computed:

$$\begin{array}{r}\overline{\mathit{add}}_{\{1\}}(y, \mathit{zero}) \to y \\ \overline{\mathit{add}}_{\{1\}}(\mathit{succ}(w), \mathit{succ}(x)) \to \mathbf{let}\ y = \overline{\mathit{add}}_{\{1\}}(w, x)\ \mathbf{in}\ y\end{array}$$

The final transformed program is thus as follows:

$$\begin{array}{r}\overline{\mathit{incL}}_{\{2\}}([\,], i) \to [\,] \\ \overline{\mathit{incL}}_{\{2\}}(w_1 : w_2, i) \to \mathbf{let}\ x = \overline{\mathit{add}}_{\{1\}}(w_1, i),\ xs = \overline{\mathit{incL}}_{\{2\}}(w_2, i)\ \mathbf{in}\ x : xs \\ \overline{\mathit{add}}_{\{1\}}(y, \mathit{zero}) \to y \\ \overline{\mathit{add}}_{\{1\}}(\mathit{succ}(w), \mathit{succ}(x)) \to \mathbf{let}\ y = \overline{\mathit{add}}_{\{1\}}(w, x)\ \mathbf{in}\ y\end{array}$$

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

[5] It proceeds from right to left in order to transform outer function calls first.

3.4 Removal of let declarations

Let expressions in transformed programs can easily be removed by applying a simplified version of *lambda lifting* [9]. In particular, we follow the transformation presented in [4, Appendix D], where a rule of the form

$$l \to \textbf{let}\ p_1 = e_1, \ldots, p_{i-1} = e_{i-1}, p_i = e_i, p_{i+1} = e_{i+1}, \ldots, p_m = e_m\ \textbf{in}\ e$$

is transformed into the rules

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

where $x_1, \ldots, x_k$ are the variables of l, $y_1, \ldots, y_m$ are the variables occurring in p_i, z is a fresh variable, and $g, g', g_1, \ldots, g_m$ 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* (see Sect. 3.2) 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 inductively sequential system is obtained:

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

3.5 Correctness

Although there exist several approaches to function inversion in the literature (e.g., [6, 8, 10, 17]), we only found a formal proof of correctness for the transformation in the work of Nishida et al. [15].

Basically, the correctness of our technique relies on [15, Theorem 9], regarding normalization and partial inversion, and the correctness of lambda lifting, regarding the removal of let expressions.

Let us first consider the work of Nishida et al. [15]. There are two kinds of differences between our method and that of [15]:

– *Restrictions.* In comparison with [15], we added several new restrictions in order to ensure that the result is “acceptable”. For instance, [15] may produce non-deterministic functions containing extra-variables which require a logical extension of reduction—called *narrowing* [18]—in order to be able to evaluate inverse functions.

– *Simplifications.* Thanks to the new restrictions, the overall method can be presented in a simpler and more intuitive way.

Obviously, the addition of new restrictions do not affect to the correctness result of [15] and, thus, Theorem 9 is still applicable.

Regarding the simplifications, they are not difficult to prove. For instance, we could easily prove that partially inverted functions are indeed inductively sequential and do not contain extra-variables; this is an immediate consequence of the preconditions in Sect. 3.1. Also, we have replaced the (more complex) unraveling of [15] by a simpler form of lambda-lifting. This is not as immediate as the above property, but could easily be proved by showing that the partial inversion of each function returns a normalized TRS. In this case, since the partial inversion is a deterministic TRS (in the terminology of [15]), then standard inlining (in most of the cases) or lambda-lifting suffices to produce a program without extra variables.

On the other hand, the correctness of the removal of let expressions is derived from the correctness of either inlining or lambda lifting [9] (see also [3, 13]), whose correctness is proved in [5]. We note that a similar transformation is considered in [15] by means of the definition of a so called *unraveling* [11].

4 Extensions of the Method

In this section, we describe how our method can be extended to cope with higher-order functions and lazy evaluation.

Higher-Order. Let us first consider a straightforward application of our method to a higher-order program. Consider, for instance, the well-known function *map*:

$$\begin{aligned} \mathit{map}(f,[\,]) &\to [\,] \\ \mathit{map}(f,x:\mathit{xs}) &\to f(x):\mathit{map}(f,\mathit{xs}) \end{aligned}$$

Then, in order to compute the partial inversion of *map* w.r.t. $\{1\}$, our method proceeds as follows. First, the normalized program is computed:

$$\begin{aligned} \mathit{map}(f,[\,]) &\to [\,] \\ \mathit{map}(f,x:\mathit{xs}) &\to \textbf{let}\ w=f(x),\ \mathit{ws}=\mathit{map}(f,\mathit{xs})\ \textbf{in}\ w:\mathit{ws} \end{aligned}$$

Now, the partial inversion step returns the following program:

$$\begin{aligned} \overline{\mathit{map}}_{\{1\}}([\,],f) &\to [\,] \\ \overline{\mathit{map}}_{\{1\}}(w:\mathit{ws},f) &\to \textbf{let}\ x=\overline{f}_{\{1\}}(w),\ \mathit{xs}=\overline{\mathit{map}}_{\{1\}}(\mathit{ws},f)\ \textbf{in}\ x:\mathit{xs} \end{aligned}$$

Finally, by removing let expressions we get

$$\begin{aligned} \overline{\mathit{map}}_{\{1\}}([\,],f) &\to [\,] \\ \overline{\mathit{map}}_{\{1\}}(w:\mathit{ws},f) &\to \overline{f}_{\{1\}}(w):\overline{\mathit{map}}_{\{1\}}(\mathit{ws},f) \end{aligned}$$

so that $\overline{map}_{\{1\}}$ maps the inverse of a function to each element of a given list. Now, the problem of how the partial inverse $\overline{f}_{\{1\}}$ can be computed arises. Since function f is not known at compile time, the pair $(f, \{1\})$ cannot be considered in the next iteration of the partial inversion algorithm.

In order to deal with such a situation, we could produce a partial inversion of the form

$$\begin{aligned} \overline{map}_{\{1\}}([\,], f) &\to [\,] \\ \overline{map}_{\{1\}}(w : ws, f) &\to inv(f, \{1\})(w) : \overline{map}_{\{1\}}(ws, f) \end{aligned}$$

where the auxiliary function *inv* is used to compute the name of the partially inverted function at run-time. In order to determine the possible values of variable f above, one could apply a standard closure analysis and/or ask the programmer. For instance, if we determine that function *map* is only called with functions *foo* and *boh*, then only $\overline{foo}_{\{1\}}$ and $\overline{boh}_{\{1\}}$ should be computed. Moreover, we should add the following definition of *inv* to the partially inverted program:

$$\begin{aligned} inv(foo, \{1\}) &\to \overline{foo}_{\{1\}} \\ inv(boh, \{1\}) &\to \overline{boh}_{\{1\}} \end{aligned}$$

Laziness. Regarding non-strict functions, our method can already be applied to lazy programs. Consider the following program:

$$\begin{aligned} foo(n, m) &\to take(n, repeat(m)) \\ take(zero, xs) &\to [\,] \\ take(succ(n), x : xs) &\to x : take(n, xs) \\ repeat(m) &\to m : repeat(m) \end{aligned}$$

where $foo(n, m)$ returns a list of n elements, all of which are m. The normalization step returns the following program:

$$\begin{aligned} foo(n, m) &\to \mathbf{let}\ x = repeat(m),\ xs = take(n, x)\ \mathbf{in}\ xs \\ take(zero, xs) &\to [\,] \\ take(succ(n), x : xs) &\to \mathbf{let}\ w = take(n, xs)\ \mathbf{in}\ x : w \\ repeat(m) &\to \mathbf{let}\ w = repeat(m)\ \mathbf{in}\ m : w \end{aligned}$$

Then, the partial inversion of *foo* w.r.t. $\{1\}$ returns

$$\overline{foo}_{\{1\}}(xs, n) \to \mathbf{let}\ m = \overline{repeat}_{\{\}}(w),\ w = \overline{take}_{\{1\}}(xs, n)\ \mathbf{in}\ m$$

Now, the next iteration computes the partial inversion of *take* w.r.t. $\{1\}$:[6]

$$\begin{aligned} \overline{take}_{\{1\}}([\,], zero) &\to xs \\ \overline{take}_{\{1\}}(x : w, succ(n)) &\to \mathbf{let}\ xs = \overline{take}_{\{1\}}(w, n)\ \mathbf{in}\ x : xs \end{aligned}$$

[6] Although the first rule violates the first precondition, we ignore this fact in this example since it is orthogonal to the kind of problem that we want to illustrate.

Therefore, the next iteration computes the partial inversion of *repeat* w.r.t. {}, and the problem shows up:

$$\overline{repeat}_{\{\}}(m : w) \to \textbf{let}\ () = \overline{repeat}_{\{1\}}(w, m)\ \textbf{in}\ m$$

In our current method, the last step suspends the partial inversion process and returns a failure because "Boolean tests" (i.e., function calls where both the arguments and the result are known) are not allowed.

The method of [15] allows the partial inversion of functions even when the result includes Boolean tests. However, observe that if one would allow such Boolean tests, we would have obtained a program like the following one (after removal of let expressions):

$$\begin{array}{rl}
\overline{foo}_{\{1\}}(xs, n) & \to \overline{repeat}_{\{\}}(\overline{take}_{\{1\}}(xs, n)) \\
\overline{take}_{\{1\}}([\,], zero) & \to xs \\
\overline{take}_{\{1\}}(x : w, succ(n)) & \to x : \overline{take}_{\{1\}}(w, n) \\
\overline{repeat}_{\{\}}(m : w) & \to m
\end{array}$$

Observe that the let expression in the right-hand side of $\overline{repeat}_{\{\}}$, i.e., the Boolean test, does not appear in the rule above because in a non-strict language its computation is not needed. Here, the meaning of function $\overline{repeat}_{\{\}}$ is as follows: given a call $\overline{repeat}_{\{\}}(xs)$, return the first element of list xs, which is clearly incorrect! (it should also check that all elements of xs are equal). This situation does not happen in our technique because the so called Boolean tests are forbidden.

5 An Inversion Tool

We have undertaken a prototype implementation of our partial inversion method in order to test its applicability and usefulness. It is implemented in Prolog (around 500 lines of code) and it is publicly available at

```
http://www.dsic.upv.es/~gvidal/german/finv/
```

Once the program is loaded into Prolog,[7] the user can load in a functional program from a file using the predicate `loadf/1`. The functional program should be written according to the following syntax for rules:

```
lhs := rhs.
```

Function and constructor symbols start with a lowercase letter and variables start with an uppercase letter (i.e., typical Prolog notation). Function definitions may also include type declarations. For instance, function *add* (see Sect. 1) can be defined as follows:

[7] Currently, it has only been tested on SWI Prolog.

```
add :: nat -> nat -> nat.

add(0,X)    := X.
add(s(X),Y) := s(add(X,Y)).
```

Arbitrary data types (like `nat` above) can also be defined by the user. For instance, natural numbers and lists can be defined as follows

```
datatype nat     ::= 0   | s(nat).
datatype list(A) ::= nil | (A : list(A)).
```

Partial inversion is then started by executing a goal of the form

```
?- invert(function_name, input_parameters_list).
```

For instance, if we type in the following goal

```
?- invert(add,[1]).
```

we get the partially inverted program:

```
add_[1](A,0)       := A.
add_[1](s(A),s(B)) := add_inv(A,B).
```

where the partial inversion of the given function is denoted by `add_[1]`.

Our preliminary results point out the viability and potential usefulness of the technique. We note, however, that one should be very careful with the election of the function and input set used for partial inversion, i.e., by choosing an arbitrary function and input set, the result is often a failure.

We also note that the current tool can only deal with first-order programs, but it could be extended to higher-order programs along the lines of the previous section. A web interface for the partial inverter can be accessed from the URL above so that the reader can easily test the system.

6 Related Work

The work by Glück and Kawabe [6] (further improved in [7]) presents an automatic program inversion algorithm for first-order functional programs. In contrast to ours, a *total* inversion algorithm is considered (a particular case of our partial inversion) and, thus, only *injective* functions produce useful results.

The closest approach is that of Nishida et al. [15], 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 the following:

– The method of [15] allows the partial inversion of functions even when the result includes “Boolean tests”. As discussed in Sect. 4, such a situation is avoided in our method in order to have a method applicable to a lazy language.

– The (more general) inversion technique of [15] introduces some additional rules that are not needed in our approach. For instance, in order to preserve the correctness, [15] adds the following rule:

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

to the definition of $\overline{add}_{\{2\}}$. These rules are not needed in our restricted method.
– Furthermore, they require a form of *narrowing* [18] to perform computations in the inverted program due to extra variables, while functional reduction suffices in our case because extra variables in partially inverted functions are not allowed.

To summarize, our method is simpler than that of [15] and can be applied in fewer cases, but when it succeeds, the resulting program is inductively sequential.

Partial inversions were also considered in [16] but, in contrast to ours, their aim is not the definition of an automatic method. Function inversion is extensively considered by Mu [14], though the author considers a different, calculational approach.

Finally, Mogensen [12] 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 might produce functions that are non-deterministic, contain extra-variables, etc., and, thus, it is not appropriate in the context of the most common functional languages. Furthermore, in contrast to ours, the semi-inversion method is rather inefficient (due to a number of non-deterministic choices); therefore, it is unclear whether an efficient implementation would be possible.

7 Discussion and Future Work

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. In contrast to other related approaches, our method is easy to implement and works well in the context of lazy evaluation.

As future work, we plan to extend the partial inversion method to cope with higher-order functions along the lines of Sect. 4. This is an interesting challenge that will allow us to design a partial inversion tool for a realistic functional programming language like Haskell.

Acknowledgements

We gratefully acknowledge the participants of IFL 2006 and the anonymous reviewers for many useful comments and suggestions.

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. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
3. O. Danvy and U.P. Schultz. Lambda-Lifting in Quadratic Time. *Journal of Functional and Logic Programming*, 2004, 2004.
4. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`.
5. A. Fischbach and J. Hannan. Specification and correctness of lambda lifting. *J. Funct. Program.*, 13(3):509–543, 2003.
6. 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.
7. 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.
8. P.G. Harrison. Function Inversion. In *Proc. of Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 153–166. North-Holland, Amsterdam, 1988.
9. 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.
10. H. Khoshnevisan and K.M. Sephton. InvX: An Automatic Function Inverter. In *In Proc. of the 3rd Int'l Conf. on Rewriting Techniques and Applications (RTA'89)*, pages 564–568. Springer LNCS 355, 1989.
11. M. Marchiori. Unraveling and Ultraproperties. In *Proc. of the 6th Int'l Conf. on Algebraic and Logic Programming (ALP'96)*, pages 107–121. Springer LNCS 1139, 1996.
12. 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.
13. M.T. Morazán and B. Mucha. Improved Graph-Based Lambda Lifting. In *Proc. of the Int'l Conf. on Software Engineering Research and Practice (SERP'06)*, pages 896–902. CSREA Press, 2006.
14. S.-C. Mu. *A Calculational Approach to Program Inversion.* PhD thesis, Oxford University Computing Laboratory, 2003.
15. 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.
16. C. Pareja-Flores and J.A. Velázquez-Iturbide. Synthesis of Functions by Transformations and Constraints. In *Proc. of the Int'l Conf. on Functional Programming (ICFP'97)*, pages 317–317. ACM Press, 1997. Poster.
17. A. Romanenko. Inversion and metacomputation. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 12–22. Sigplan Notices, 26(9), ACM, New York, 1991.
18. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.
19. P. Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proc. of 14th ACM Symp. on Principles of Programming Languages (POPL'87)*, pages 307–313. ACM Press, 1987.