

# 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). To ease the definition, our method proceeds in a stepwise manner: normalisation (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 quite 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* [16], 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

$$f^{-1}(t) = \langle t_1, \dots, t_n \rangle$$

if and only if

$$f(t_1, \dots, t_n) = t$$

for all terms  $t_1, \dots, t_n, t$ . 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).

---

<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.

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

$$\bar{f}_I(t, t_{i_1}, \dots, t_{i_m}) = \langle t_{j_1}, \dots, t_{j_k} \rangle$$

if and only if

$$f(t_1, \dots, t_n) = t$$

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 (when  $I = \emptyset$ ). 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*):

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

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:

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

Their definitions can be given, respectively, as follows:

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

Observe that both  $\overline{\text{add}}_{\{1\}}$  and  $\overline{\text{add}}_{\{2\}}$  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 inverses,

- 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, 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  $\overline{add}_{\emptyset}$  is not so obvious.

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

- The method proceeds in a stepwise manner: normalisation (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.

The paper is organised as follows. After introducing some preliminaries in the next section, we present in Sect. 3 our method for partial inversion. Section 4 describes the implementation of our inversion tool. Finally, Sect. 5 discusses some related work and Sect. 6 concludes.

## 2 Preliminaries

We follow the standard framework of *term rewriting* [4] for developing our results since it suffices to model the first-order component of many functional programming languages.

### 2.1 Term Rewriting Systems

In term rewriting, 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 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  $\mathcal{F} \cap \mathcal{V} = \emptyset$ .

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*  $o_1, \dots, o_n$ .

The evaluation of terms w.r.t. a TRS is formalised 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.

## 2.2 Inductively Sequential Systems

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).

Two (possibly renamed) constructor-based rules  $l \rightarrow r$  and  $l' \rightarrow r'$  *overlap* if there exists a unifier of  $l$  and  $l'$ , i.e., a substitution  $\sigma$  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. The formal definition of this class of programs requires the notion of

*definitional tree* (originally introduced in [1], though we follow the definition of [2] which is more appropriate for our purposes).

A linear term of the form  $f(t_1, \dots, t_n)$  with  $f/n \in \mathcal{D}$  a function symbol and  $t_1, \dots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$  constructor terms is called a *lhs-term* (since this is the shape of the terms in the left-hand sides of the rules).

A *definitional tree* of a finite set  $S$  of lhs-terms is a non-empty set  $\mathcal{P}$  of lhs-terms partially ordered by subsumption<sup>3</sup> having the following properties:

*Root property:*  $\mathcal{P}$  has a minimum element (that we denote as  $root(\mathcal{P})$ ), i.e., a term that is more general than any other term in  $\mathcal{P}$ .

*Leaves property:* The maximal elements of  $\mathcal{P}$ , called the *leaves* of the definitional tree, are the elements of  $S$ . Non-maximal elements are also called *branch nodes*.

*Parent property:* If  $\pi \in \mathcal{P}$ ,  $\pi \neq root(\mathcal{P})$ , there exists a unique  $\pi' \in \mathcal{P}$ , called the *parent* of  $\pi$  (and  $\pi$  is called a *child* of  $\pi'$ ), such that  $\pi' < \pi$  and there is no other pattern  $\pi'' \in \mathcal{T}(\mathcal{D}, \mathcal{V})$  with  $\pi' < \pi'' < \pi$ .

*Induction property:* Given  $\pi \in \mathcal{P} \setminus S$ , there is a position  $o$  in  $\pi$  with  $\pi|_o \in \mathcal{V}$  (called the *inductive position*), and constructors  $c_1/k_1, \dots, c_n/k_n \in \mathcal{C}$  with  $c_i \neq c_j$  for  $i \neq j$ , such that, for all  $\pi_1, \dots, \pi_n$  which have the parent  $\pi$ ,  $\pi_i = \pi[c_i(\overline{x_{k_i}})]_o$  (where  $\overline{x_{k_i}}$  are new distinct variables) for all  $1 \leq i \leq n$ .

If  $\mathcal{R}$  is an orthogonal TRS and  $f/n$  is a defined function, we call  $\mathcal{P}$  a *definitional tree of  $f$*  if  $root(\mathcal{P}) = f(\overline{x_n})$  for distinct variables  $\overline{x_n}$  and the leaves of  $\mathcal{P}$  are all (and only) variants of the left-hand sides of the rules in  $\mathcal{R}$  defining  $f$ . A defined function is called *inductively sequential* if it has a definitional tree. A rewrite system  $\mathcal{R}$  is called *inductively sequential* if all its defined functions are inductively sequential.

It is often convenient and simplifies understanding to provide a graphic representation of definitional trees, where the inductive position in branch nodes is surrounded by a box and the leaves contain the corresponding rules.

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

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

This function is inductively sequential because there exists a definitional tree  $\mathcal{P}$ :

$$\mathcal{P} = \{n \leq m, \text{ zero } \leq y, \text{ succ}(x) \leq y, \text{ succ}(x) \leq zero, \text{ succ}(x) \leq succ(y)\}$$

which includes the left-hand sides of the functions. The graphic representation of  $\mathcal{P}$  is then as follows:

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

<sup>3</sup> A term  $s$  *subsumes* (or is *more general* than) a term  $t$ , in symbols  $s \leq t$ , iff there exists a substitution  $\sigma$  such that  $\sigma(s) = t$ . Also, we have  $s < t$  if  $s \leq t$  holds but  $t \leq s$  does not.

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, 3].

### 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  $I \subset \{1, \dots, n\}$  of input (or “known”) parameters. Therefore, we want to obtain a new function, which we call  $\bar{f}_I$ , 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  $\bar{I} = \{1, \dots, n\} \setminus I$  (the “unknown” parameters).

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 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  $I = \{i_1, \dots, i_m\} \subset \{1, \dots, n\}$  iff the following conditions hold:*

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

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

As mentioned before, the first condition above is often ignored (e.g., [13]), but we require it in order to produce useful programs 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.  $I \subset \{1, \dots, n\}$ . Assume a relation  $Rel(f)$ , defined as follows:

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

where  $t_1, \dots, t_n, t$  are ground constructor terms (i.e., *values*). Then, the partial inverse of  $f$  w.r.t.  $I$  is well-defined when there are no tuples  $(t_1, \dots, t_n, t)$  and  $(s_1, \dots, s_n, s)$  in  $Rel(f)$  such that  $(t_{i_1}, \dots, t_{i_m}, t) = (s_{i_1}, \dots, s_{i_m}, s)$  and  $(t_{j_1}, \dots, t_{j_k}) \neq (s_{j_1}, \dots, s_{j_k})$ , where  $\bar{I} = \{j_1, \dots, j_k\}$ . Observe that this condition amounts to injectivity when  $I = \emptyset$ , but is more relaxed when  $I \neq \emptyset$ .

Unfortunately, this condition is generally undecidable. Therefore, we introduce three (decidable) preconditions for partial inversion. The first precondition, which regards extra variables, is very simple:

**Definition 2 (first precondition: extra variables).** *Let  $f/n$  be a function to be partially inverted w.r.t.  $I = \{i_1, \dots, i_m\} \subset \{1, \dots, n\}$ . Then, function  $f/n$  must fulfil the following condition:  $Var(\{t_{j_1}, \dots, t_{j_k}\}) \subseteq Var(\{r, t_{i_1}, \dots, t_{i_m}\})$  for every rule  $f(t_1, \dots, t_n) \rightarrow r$  in the definition of  $f$ , with  $\bar{I} = \{j_1, \dots, j_k\}$ .*

For instance, a function  $fst$  defined by a rule of the form

$$fst(x, y) \rightarrow x$$

cannot be partially inverted w.r.t.  $\{1\}$  since  $Var(\{y\}) \not\subseteq Var(\{x, x\})$ . Indeed, the definition of the partially inverted function  $\overline{fst}_{\{1\}}$  would contain an extra variable:

$$\overline{fst}_{\{1\}}(x, x) \rightarrow y$$

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 second precondition regards left-linearity and is also rather simple:

**Definition 3 (second precondition: left-linearity).** *Let  $f/n$  be a function to be partially inverted w.r.t.  $I = \{i_1, \dots, i_m\} \subset \{1, \dots, n\}$ . Then,  $C$  must be linear and do not share variables with  $t_{i_1}, \dots, t_{i_m}$  for every rule  $f(t_1, \dots, t_n) \rightarrow C[e_1, \dots, e_l]$  in the definition of function  $f$ .*

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

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

where lists are built from  $[\ ]$  and “:”. This function does not fulfil 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.  $\emptyset$  would return the following rules:

$$\begin{aligned} \overline{double}_{\emptyset}([\ ]) &\rightarrow [\ ] \\ \overline{double}_{\emptyset}(x : x : xs) &\rightarrow x : \overline{double}_{\emptyset}(xs) \end{aligned}$$

Also, the partial inversion of function  $fst$  w.r.t.  $\{1\}$  above does not fulfil 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  $\overline{double}_{\emptyset}$  could be transformed as follows:

$$\frac{\overline{double}_{\emptyset}([]) \rightarrow []}{\overline{double}_{\emptyset}(x : y : xs) \rightarrow cond(eq(x, y), x : \overline{double}_{\emptyset}(xs))}$$

where  $cond(c, t)$  returns  $t$  if  $c$  evaluates to *true* and  $eq(t1, t2)$  is a Boolean equality test. Such a transformation, however, would not be useful in a lazy context because  $eq$  can 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.

**Definition 4 (third precondition: inductive sequentiality).** *Let  $f/n$  be a function to be partially inverted w.r.t.  $I = \{i_1, \dots, i_m\} \subset \{1, \dots, n\}$ . Then, there must be a definitional tree for a function  $\overline{f}_I$  whose definition contains the following left-hand sides:*

$$\left\{ \overline{f}_I(C[x_1, \dots, x_l], t_{i_1}, \dots, t_{i_m}) \mid f(t_1, \dots, t_n) \rightarrow C[e_1, \dots, e_l] \in \mathcal{R}_f \text{ and } x_1, \dots, x_l \text{ are fresh variables} \right\}$$

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  $app$ :

$$\begin{aligned} app([], y) &\rightarrow y \\ app(x : xs, y) &\rightarrow x : app(xs, y) \end{aligned}$$

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).

In the remainder of this section, we present our stepwise process for partial inversion.

### 3.2 Normalisation

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 5 (normalized TRS).** *A normalised TRS contains either 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 (i.e., nested defined function symbols are not allowed). Each equality,  $p_i = e_i$ , is called a pattern definition. We further require that  $\mathcal{V}ar(e_i) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \dots \cup \mathcal{V}ar(p_{i-1})$ , for  $i = 1, \dots, n$ , and  $\mathcal{V}ar(p_0) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(p_1) \cup \dots \cup \mathcal{V}ar(p_n)$ .<sup>4</sup>

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 normalisation process:

**Definition 6 (normalization).** *Given a TRS  $\mathcal{R}$ , the normalised 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:*

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

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, normalisation proceeds as follows: if the right-hand side is a constructor term, then it is already normalised; 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, normalisation replaces those operation-rooted subterms by fresh variables and adds new pattern definitions by means of a let declaration.

<sup>4</sup> This is similar to the notion of *deterministic* conditional TRS.

- 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 fulfil the condition on the variables of Def. 5.

Observe that every normalised 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 normalised TRSs as much as possible:

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

Note that these rules are well-defined in our case because patterns  $p_i$  are always variables by Def. 6. 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:

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

The normalisation of this program returns

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

### 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:

- The algorithm takes a normalised program and returns either a failure or a normalised 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  $I \subset \{1, \dots, n\}$ .
- 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. 7).
- 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  $\overline{g}_J(t_1, \dots, t_m)$  in these right-hand sides.

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

**Fig. 1.** Partial inversion algorithm

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

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

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

**Definition 7 (function  $pinv$ ).** Given a normalised TRS  $\mathcal{R}$ , a function  $f/n$ , and a set  $I \subset \{1, \dots, n\}$ , the partial inversion of  $f$  w.r.t.  $I$ , in symbols  $pinv(\mathcal{R}, f, I)$ , is obtained as the set

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

Function  $\llbracket \cdot \rrbracket$  is defined as follows:

$$\begin{aligned}
\llbracket f(\overline{p_n}) \rightarrow C \rrbracket_I &= \overline{f}_I(C \llbracket \cdot \rrbracket, p_{i_1}, \dots, p_{i_m}) \rightarrow \langle p_{j_1}, \dots, p_{j_k} \rangle \\
\llbracket f(\overline{p_n}) \rightarrow \mathbf{let} \overline{q_l} = \overline{e_l} \mathbf{in} C \rrbracket_I &= \overline{f}_I(C \llbracket \cdot \rrbracket, p_{i_1}, \dots, p_{i_m}) \rightarrow ((\mathbf{let} \overline{q_l} = \overline{e_l} \\
&\quad \mathbf{in} \langle p_{j_1}, \dots, p_{j_k} \rangle))_V^l
\end{aligned}$$

where  $I = \{i_1, \dots, i_m\}$ ,  $\bar{I} = \{j_1, \dots, j_k\}$ , and  $V = \text{Var}(\bar{f}_I(C[], p_{i_1}, \dots, p_{i_m}))$ . The auxiliary function  $(\bar{\phantom{x}})$  is defined inductively as shown in Fig. 2.

Essentially, function  $\text{pinv}$  above considers sequentially<sup>5</sup> 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 initialised to the variables of the new left-hand side) as follows:

- If all variables in  $q_1, \dots, q_b$  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  $\{i_1, \dots, i_m\}$  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  $\{j_1, \dots, j_k\}$ , and replace the original pattern definition by  $\langle q_{j_1}, \dots, q_{j_k} \rangle = \bar{g}_{\{i_1, \dots, i_m\}}(p_l, q_{i_1}, \dots, q_{i_m})$ .

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

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

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

where

$$\begin{aligned} & ((\text{let } w_1 = \text{add}(i, x), w_2 = \text{incL}(xs, i) \text{ in } x : xs))_{\{w_1, w_2, i\}}^2 \\ &= ((\text{let } w_1 = \text{add}(i, x), xs = \overline{\text{incL}}_{\{2\}}(w_2, i) \text{ in } x : xs))_{\{w_1, w_2, i, xs\}}^1 \\ &= ((\text{let } x = \overline{\text{add}}_{\{1\}}(w_1, i), xs = \overline{\text{incL}}_{\{2\}}(w_2, i) \text{ in } x : xs))_{\{w_1, w_2, i, xs, x\}}^0 \\ &= \text{let } x = \overline{\text{add}}_{\{1\}}(w_1, i), xs = \overline{\text{incL}}_{\{2\}}(w_2, i) \text{ in } x : xs \end{aligned}$$

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

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

<sup>5</sup> It proceeds from right to left in order to transform outer function calls first.



### 3.5 Correctness

Although there exist several approaches to function inversion in the literature (e.g., [6, 8, 10, 14]), we only found a formal proof of correctness for the transformation in the work of Nishida et al. [13].

The correctness of our transformation can also be derived from the results in [13] for the partial inversion of term rewriting systems. On the one hand, the first two steps: normalisation and partial inversion, are essentially equivalent (though simpler) to their inversion transformation; there are several differences though, but they mainly restrict the kind of programs that can be transformed in order to get useful results (see Sect. 5). Furthermore, our normalised TRSs can be regarded as deterministic conditional TRSs of type 3 (the output of the transformation of Nishida et al). Therefore, Theorem 9 in [13] can also be used to prove the correctness of the first two steps of our transformation.

On the other hand, the correctness of the elimination of let expressions by either inlining or lambda lifting is well known. A similar transformation is considered in [13] by means of the definition of a so called *unraveling* [11].

## 4 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,<sup>6</sup> 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 *app* (see Sect. 3.1) can be defined as follows:

```
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)).
```

---

<sup>6</sup> Currently, it has only been tested on SWI Prolog.

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_inv(A,0)      =: A.
add_inv(s(A),s(B)) =: add_inv(A,B).
```

where the partial inversion of the given function is denoted by `add_inv` and the symbol “:=” is replaced by “=:” in the rules.

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 (i.e., a TRS that is not inductively sequential).

A web interface for the partial inverter can be accessed from the URL above so that the reader can easily test the system.

## 5 Related Work

Among the closest approaches, we have the work by Glück and Kawabe [6] (further improved in [7]), 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. [13], 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 [13] allows the partial inversion of functions even when the result includes “Boolean tests”. For instance, given the following function:

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

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

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

Here, the pattern definition  $\langle \rangle = \overline{from}_{\{1\}}(y, succ(x))$  is only used to test that  $y$  is a list headed by  $succ(x)$ . 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  $from$  (i.e., what we call a Boolean test). Therefore, we would return a failure.

- The (more general) inversion technique of [13] introduces some additional rules that are not needed in our approach. For instance, in order to preserve the correctness, [13] 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* [15] 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 summarise, our method is simpler than that of [13] and can be applied in fewer cases, but when it succeeds, the resulting program is inductively sequential.

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 the underlying techniques are also similar to those of [6, 13] and ours.

## 6 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. 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 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. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of the Int'l Conference on Algebraic and Logic Programming, ALP'97*, pages 16–30. Springer LNCS 1298, 1997.
3. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.
4. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
5. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: <http://www.informatik.uni-kiel.de/~mh/curry/>.
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 the 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. 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.
14. A. Romanenko. Inversion and metacomputation. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 12–22. Sigplan Notices, 26(9), ACM, New York, 1991.
15. J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.
16. 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.