Institute of Electronics, Information, and Communication Engineers

一般社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2012-27 KBSE2012-29 (2012-7)

On Extending Matching Operation in Grammar Programs for Program Inversion

丹羽 南† 西田 直樹†† 酒井 正彦†† 坂部 俊樹†† 草刈圭一朗††

†,†† 名古屋大学大学院情報科学研究科
〒 464-8603 名古屋市千種区不老町
E-mail: †mniwa@sakabe.i.is.nagoya-u.ac.jp, ††{nishida,sakai,sakabe,kusakari}@is.nagoya-u.ac.jp

On Extending Matching Operation in Grammar Programs for Program Inversion

Minami NIWA†, Naoki NISHIDA††, Masahiko SAKAI††,
Toshiki SAKABE††, and Keiichirou KUSAKARI††

†,†† Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
E-mail: †mniwa@sakabe.i.is.nagoya-u.ac.jp, ††{nishida,sakai,sakabe,kusakari}@is.nagoya-u.ac.jp

Abstract The inversion method proposed by Glück and Kawabe uses grammar programs as intermediate results, that comprise sequences of operations (data generation, matching, etc.). The semantics of the grammar programs is defined as a stack machine where data terms are stored in the stack. The matching operation for a constructor symbol pops up the topmost term of the stack and pushes its direct subterms to the stack if the topmost term is rooted by the constructor. This paper strengthens the matching operation by augmenting a parameter that specifies what position the matching operation can look into from the top of the stack. This extension is sometimes effective to avoid the failure execution of the determinization method based on LR parsing described in the latter half of the inversion method.

Key words program inversion, term rewriting systems, LR parsing

1. Introduction

Inverse computation for an n-ary functions f is, given an output v of f, the calculation of (all) the possible inputs $v_1, \ldots, v_n$ of f such that $f(v_1, \ldots, v_n) = v$. As an approach to inverse computation, *program inversion* have been studied [1–5, 7–9, 11, 12], which take the definition of f as input and compute the definition of the inverse function f^{-1}. Surprisingly, the essential ideas of the existing methods for inversion are almost the same, except for [5, 9], and inverse programs generated to the methods are in general non-deterministic with respect to the application of function definitions even if the target functions are injective. For this reason, determinization of such inverse programs is one of the most interesting topics in developing inversion methods and has been investigated in several ways [2–4, 6].

The inversion method LRinv, proposed by Glück and Kawabe [2–4], adopts an interesting approach to determinization. A functional program is once translated into a context-free grammar, called a *grammar program*, whose language is consistent with all possible execution sequences of operations (data generation, matching, etc.)—the language is semantically equivalent to the given functional program. The semantics of the operations is defined as a stack machine where data terms are stored in the stack (input arguments, intermediate results, and outputs), in which the meanings of operations are defined as *pop* and *push* operations on the stack. The grammar program is inverted by reversing context free grammars and replacing each operation by its opposite operation, then it is determinized by a method based on *LR(0) parsing* into a grammar program, which is easily convertible to a functional program without overlapping between function

This article is a technical report without peer review, and its polished and/or extended version may be published elsewhere.
Copyright ©2012 by IEICE
NII-Electronic Library Service

Institute of Electronics, Information, and Communication Engineers

definitions. The mechanism of the determinization is complicated but quite interesting, e.g., some grammar program corresponding to non-terminating and overlapping function definitions is transformed into the grammar program corresponding to a terminating and non-overlapping function definitions. Moreover, the determinization method is independent from the inversion principle, and thus, it is useful as a post-process of any other inversion methods. However, the determinization is not successful for all grammar programs obtained by the inversion phase; there exist some examples for which this method fails but another inversion method succeeds (cf. [9]). The failure of the determinization is mainly caused from *shift/shift conflicts* that are branches in the collection of *item sets* with two or more different atomic operations, except for matching ones. In this determinization, grammar programs with such conflicts do not proceed to the code generation phase.

In this paper, we try to avoid shift/shift conflicts as much as possible, applying the code generation method to grammar programs. To this end, we extend the syntax and semantics of matching operations in grammar programs. In the abstract semantics of programs, any value used in the execution is freely accessed by means of variable names. The original matching operation for a constructor pops the topmost element from the stack and pushes direct subterms of the element to the stack if the topmost term is rooted by the constructor. This means that the operation refers to the topmost element of the stack only, and thus, the permutation of the elements in the stack is often necessary to access values addressed at non-topmost of the stack. To avoid the use of such permutations for matching operations, we extend matching operations by specifying natural number k so as to allow them to take the kth topmost element of the stack. This extension sometimes avoids the failure execution caused by the shift/shift conflict, and thus, allows us to proceed to the code generation phase. We show some examples for which the original method is not successful but the extended one succeeds in producing non-overlapping programs. We also show an unsuccessful example for which the extended method produces an overlapping and terminating program, while the original method fails.

The contribution of this paper is to make LRinv strictly more powerful.

This paper is organized as follows. In Section 2., we briefly recall LRinv by means of an example. In Section 3., we extend the syntax and semantics of matching operations in grammar programs, and show a successful examples and an unsuccessful example. In Section 4., we briefly describe future work of this research.

2. Overview of LRinv

In this section, we briefly recall LRinv [2–4] by using an example, while we describe the complete syntax and semantics of grammar programs. We also recall the concrete definitions of item sets and conflicts related to item sets, called *shift/shift conflicts*, that influence the success or failure of the determinization method (i.e., LRinv).

LRinv assumes that functions to be inverted are injective over inductive data structures. Throughout this paper, we use $\mathcal{C}$ and $\mathcal{F}$ for finite sets of *constructor* and *function* symbols that appears in programs, where each symbol f has a fixed arity n, represented by f/n. The set of ground terms over $\mathcal{C}$ is denoted by $T(\mathcal{C})$. For the sake of readability, functional programs are written as *constructor TRSs* (cf., [10]). The syntax of grammar programs is defined by the following BNF:(1)

$$
\begin{array}{lll}
q ::= d_1 \ldots d_m & & \text{(grammar program)} \\
d ::= f \to t_1 \ldots t_m & & \text{(definition)} \\
t ::= a & & \text{(atomic operation)} \\
\quad \mid f & & \text{(function call)} \\
a ::= c! & & \text{(constructor application)} \\
\quad \mid c? & & \text{(pattern matching)} \\
\quad \mid \pi & & \text{(permutation)} \\
\pi ::= (i_1 \ldots i_n) & &
\end{array}
$$

where $f \in \mathcal{F}$, $c \in \mathcal{C}$, $m \geqq 0$, $n > 0$, $\{i_1, \ldots, i_n\} = \{1, \ldots, n\}$. Note that function symbols are *non-terminals* and others are *terminals*.(2) The set of grammar programs is denoted by $\mathcal{G}$. For a grammar program $G \in \mathcal{G}$ defining a function $f \in \mathcal{F}$, a notation $L(G, f)$ represents the set of operation sequences generated from the non-terminal f, where operations are function calls and atomic operations. The set of operations is denoted by $\mathcal{O}p$ and the set of atomic operations is denoted by $\mathcal{A}$. To represent sequences of objects, we may use the usual list constructor : and the empty list ϵ, e.g., the sequence 1 2 3 is denoted by $1 : 2 : 3 : \epsilon$. We may write @ as the binary operator for list concatenation.

The semantics $\to_G$ of a grammar program $G \in \mathcal{G}$ is defined over pairs of sequences operations and terms, a binary

(1): For the sake of readability, this paper does not deal with the *duplication/equality* operator $\lfloor\_\rfloor$.

(2): For an operation t, $\mathrm{pop}(t)$ and $\mathrm{push}(t)$ denote the numbers of values popped and pushed by t, resp.: $\mathrm{pop}(f) = n$ and $\mathrm{push}(f) = m$ for an n-ary function $f \in \mathcal{F}$ returning m values; $\mathrm{pop}(c?) = \mathrm{push}(c!) = 1$ and $\mathrm{push}(c?) = \mathrm{pop}(c!) = n$ for an n-ary constructor $c \in \mathcal{C}$; $\mathrm{pop}(\pi) = \mathrm{push}(\pi) = n$ for a permutation π with the length n. Note that $f \to t_1 \ldots t_m$ should be balanced, i.e., $\mathrm{pop}(f) + \sum_{j=1}^{m}(\mathrm{push}(t_j) - \mathrm{pop}(t_j)) = \mathrm{push}(f)$ and $\mathrm{pop}(f) + \sum_{j=n}^{k-1}(\mathrm{push}(t_j) - \mathrm{pop}(t_j)) \geqq \mathrm{pop}(t_k)$ for all $1 \leqq k \leqq m-1$. These (in)equalities are necessary to guess the arity and coarity of general auxiliary functions, and thus, useful to translate grammar programs back into functional programs.

NII-Electronic Library Service

Institute of Electronics, Information, and Communication Engineers

relation over $\mathcal{O}p^* \times T(\mathcal{C})^*$, as follows:

$$\begin{aligned}
(f : ts, vs) &\to_G (us@ts, vs) \\
&\quad \text{where } f \to us \in G \\
(c! : ts, v_1 : \cdots : v_n : vs) &\to_G (ts, c(v_1, \ldots, v_n) : vs) \\
&\quad \text{where } c/n \in \mathcal{C} \\
(c? : ts, c(v_1, \ldots, v_n) : vs) &\to_G (ts, v_1 : \cdots : v_n : vs) \\
&\quad \text{where } c/n \in \mathcal{C} \\
((i_1 \ \ldots \ i_n) : ts, v_1 : \cdots : v_n : vs) &\to_G (ts, v_{i_1} : \cdots : v_{i_n} : vs)
\end{aligned}$$

Let f be an n-ary function that returns a tuple of m values. When computing $f(v_1, \ldots, v_n)$, we start with $(f : \epsilon, v_1 : \cdots : v_n : \epsilon)$ and obtain the result $(u_1, \ldots, u_m)$ if $(f : \epsilon, v_1 : \cdots : v_n : \epsilon) \to_G^* (\epsilon, u_1 : \cdots : u_m : \epsilon)$. The second component of the pairs plays a role of a *stack*. Let c be an n-ary constructor symbol. The constructor application $c!$ pops the n topmost values $v_1, \ldots, v_n$ from the stack and pushes the value $c(v_1, \ldots, v_n)$ onto the stack. The pattern matching $c?$ pops the topmost value $c(v_1, \cdots, v_n)$ from the stack and then pushes values $v_1, \ldots, v_n$ onto the stack. A permutation $(i_1 \ \ldots \ i_n)$ reorders the n topmost values on the stack by moving in parallel the i_jth element to the jth position. We merge sequences of permutations in G into single optimized ones, such as (2 1) (3 2 1) into (3 2 1), and remove identity permutations, such as (1 2 3).

Example 1. Let us consider the function snoc defined by the following TRS:

$$\left\{ \begin{array}{c} \mathsf{snoc}(\mathsf{nil}, y) \to \mathsf{cons}(y, \mathsf{nil}) \\ \mathsf{snoc}(\mathsf{cons}(x, xs), y) \to \mathsf{cons}(x, \mathsf{snoc}(xs, y)) \end{array} \right\}$$

where $\mathcal{C} = \{\mathsf{nil}/0, \mathsf{cons}/2, \ldots\}$ and $\mathcal{F} = \{\mathsf{snoc}/2\}$. The function snoc appends the second argument to the end of the first argument, e.g., $\mathsf{snoc}(\mathsf{cons}(1, \mathsf{cons}(2, \mathsf{nil})), 3) = \mathsf{cons}(1, \mathsf{cons}(2, \mathsf{cons}(3, \mathsf{nil})))$. This program is translated into the following grammar program:

$$G_{\mathsf{snoc}} = \left\{ \begin{array}{l} \mathsf{snoc} \to \mathsf{nil?}\ \mathsf{nil!}\ (2\ 1)\ \mathsf{cons!} \\ \mathsf{snoc} \to \mathsf{cons?}\ (2\ 3\ 1)\ \mathsf{snoc}\ (2\ 1)\ \mathsf{cons!} \end{array} \right\}$$

The grammar program G_{snoc} is inverted as follows:

$$G_{\mathsf{snoc}^{-1}} = \left\{ \begin{array}{l} \mathsf{snoc}^{-1} \to \mathsf{cons?}\ (2\ 1)\ \mathsf{nil?}\ \mathsf{nil!} \\ \mathsf{snoc}^{-1} \to \mathsf{cons?}\ (2\ 1)\ \mathsf{snoc}^{-1}\ (3\ 1\ 2)\ \mathsf{cons!} \end{array} \right\}$$

Note that this TRS is also obtained by another inversion method, e.g., [9]. If the inverse TRS resulting from this simple inversion above is non-overlapping, program inversion is completed. However, $G_{\mathsf{snoc}^{-1}}$ does not correspond to a non-overlapping TRS (see $R_{\mathsf{snoc}^{-1}}$). Thus, we proceed to the determinization method based LR(0) parsing as follows.

Collecting item sets. Given a grammar program, the collection of LR(0) items sets is computed by a closure operation (see below).

Code generation. Given a *conflict-free* collection of LR(0) item sets, a grammar program of which the corresponding TRS is non-overlapping is generated. For lack of space, we do not describe the detail.

In the following, we recall how to construct from a grammar program the canonical LR(0) collection. An *LR(0) parse item* (an *item*, for short) is a function definition with a dot "$\cdot$" at some position on the right side. An item indicates how much of a sequence of operations has been performed at a certain point during the evaluation of a grammar program. We group items together into *LR(0) item sets* (*item sets*, for short) which represent the set of all possible operations that a computation can take at a certain point during evaluation. For a grammar program q, to calculate the collection $\mathcal{I}$ of all reachable item sets, we introduce two relations, *shift* and *reduce*, which correspond to determining the parse action in LR(0) parsing:

(shift) $I_1 \rightsquigarrow^t I_2$ iff $I_2 = \{f \to ts_1\ t \cdot ts_2 \mid f \to ts_1 \cdot t\ ts_2 \in closure(I_1)\}$

(reduce) $I \hookrightarrow f$ iff $f \to t_1 \ldots t_n \cdot \in closure(I)$

where the function $closure : \mathcal{I} \to \mathcal{I}$ and its auxiliary function $cls : \mathcal{I} \times \mathcal{F} \to \mathcal{I}$ are defined as

- $closure(I) = cls(I, \emptyset)$
- $cls(\emptyset, F) = \emptyset$
- $cls(I, F) = I \cup cls(\{f \to \cdot\ ts \mid f \to ts \in q, f \in F'\}, F \cup F')$ where $F' = \{f \mid f' \to ts_1 \cdot f\ ts_2 \in I, f \notin F\}$

Shift with an operation t transforms item set I_1 to item set I_2 under t, reduce is to return from item set I after n operations of function f were shifted, and $closure(I)$ calculates a new item set I of grammar program q. Intuitively, item $f' \to ts_1 \cdot t\ ts_2 \in closure(I)$ indicates that, at some point during evaluation of program q, we may perform operation t. The closure calculation terminates since there is only a finite number of different items for every program.

The set of all item sets of a program q, the *canonical collection* $\mathcal{I}_q$, is defined as follows:

- $\mathcal{I}_q = \{I \mid I_0 \rightsquigarrow^* I\}$ where $I_0 = \{\mathsf{s} \to \cdot\ h\}$ and $h = main(q)$
- $I_1 \rightsquigarrow^* I_2$ iff $I_1 = I_2 \ \vee\ (\exists t.\ \exists I'.\ I_1 \rightsquigarrow^t I' \ \wedge\ I' \rightsquigarrow^* I_2)$

where, $main(q)$ denotes the main function of q, s is a new function symbol. The set is finite since there is only a finite number of different item set.

At the end of Step 1 (collecting item set), we examine whether there is a conflict in the collection or not: if there is no conflict, we proceed to Step 2 (code generation). An item set I is said to be *conflict-free* if it does not have any of the following three conflicts:

(shift/reduce) $I \rightsquigarrow^t I' \wedge I \hookrightarrow^n f$

(reduce/reduce) $I \hookrightarrow^{n_1} f_1 \wedge I \hookrightarrow^{n_2} f_2 \wedge (f_1, n_1) \neq (f_2, n_2)$

(shift/shift) $I \rightsquigarrow^{a_1} I_1 \wedge I \rightsquigarrow^{a_2} I_2 \wedge \{a_1, a_2\} \not\subseteq \{c? \mid c \in \mathcal{C}\}$

NII-Electronic Library Service

Institute of Electronics, Information, and Communication Engineers

In this paper, we focus on *shift/shift conflicts* only since such conflicts are much more important for the determinization method. If an item set has two or more shift actions labeled with atomic operations, all of them must be matching operations. Shift/shift conflict is the case that an item set has two or more shift actions labeled with atomic operations whose one or more is not matching operations. For instance, $I \rightsquigarrow^{\mathsf{nil!}} I'$ and $I \rightsquigarrow^{\mathsf{cons?}} I''$ are in shift/shift conflicts, while $I \rightsquigarrow^{\mathsf{nil?}} I'$ and $I \rightsquigarrow^{\mathsf{cons?}} I''$ are not.

Example 2. Consider the grammar program $G_{\mathsf{snoc}^{-1}}$ in Example 1 again. $G_{\mathsf{snoc}^{-1}}$ is conflict-free, we proceeds to Step 2, transforming the grammar program $G_{\mathsf{snoc}^{-1}}$ into the following grammar program:

$$\left\{\begin{array}{l} \mathsf{snoc}^{-1} \to \mathsf{cons?}\ (2\ 1)\ \mathsf{f} \\ \quad \mathsf{f} \to \mathsf{nil?}\ \mathsf{nil!} \\ \quad \mathsf{f} \to \mathsf{cons?}\ (2\ 1)\ \mathsf{f}\ (3\ 1\ 2)\ \mathsf{cons!} \end{array}\right\}$$

where f is a fresh non-terminal symbol introduced by means of the determinization. Considering f as an auxiliary function of snoc, this resulting grammar program is translated back into the following non-overlapping conditional TRS:

$$\left\{\begin{array}{rl} \mathsf{snoc}^{-1}(\mathsf{cons}(x, xs)) & \to \mathsf{f}(xs, x) \\ \mathsf{f}(\mathsf{nil}, y) & \to (\mathsf{nil}, y) \\ \mathsf{f}(\mathsf{cons}(x, xs), y) & \to (\mathsf{cons}(y, z), w) \\ & \Leftarrow \mathsf{f}(xs, x) \twoheadrightarrow (z, w) \end{array}\right\}$$

Next, let us consider an unsuccessful example.

Example 3. Consider the following constructor TRS:

$$\left\{\begin{array}{rl} \mathsf{unbin2}(x) & \to \mathsf{ub}(x, \mathsf{nil}) \\ \mathsf{ub}(\mathsf{suc}(\mathsf{zero}), y) & \to y \\ \mathsf{ub}(\mathsf{suc}(\mathsf{suc}(\mathsf{x})), y) & \to \mathsf{ub}(\mathsf{suc}(x), \mathsf{inc}(y)) \\ \mathsf{inc}(\mathsf{nil}) & \to \mathsf{cons}(0, \mathsf{nil}) \\ \mathsf{inc}(\mathsf{cons}(0, xs)) & \to \mathsf{cons}(1, xs) \\ \mathsf{inc}(\mathsf{cons}(1, xs)) & \to \mathsf{cons}(0, (\mathsf{inc}(xs))) \end{array}\right\}$$

where $\mathcal{C} = \{\mathsf{nil}/0, \mathsf{cons}/2, 0/0, 1/0, \mathsf{zero}/0, \mathsf{suc}/1, \ldots\}$ and $\mathcal{F} = \{\mathsf{unbin2}/1, \mathsf{ub}/2, \mathsf{inc}/1\}$. The function unbin2 converts natural numbers represented as suc(zero), suc(suc(zero)),... to binary-numeral expressions nil, cons(0, nil),....(3) This program is translated and inverted to the following grammar program:

$$G_{\mathsf{unbin2}^{-1}} = \left\{\begin{array}{l} \mathsf{unbin2}^{-1} \to \mathsf{ub}^{-1}\ (2\ 1)\ \mathsf{nil?} \\ \quad \mathsf{ub}^{-1} \to \mathsf{zero!}\ \mathsf{suc!} \\ \quad \mathsf{ub}^{-1} \to \mathsf{ub}^{-1}\ \mathsf{suc?}\ (2\ 1)\ \mathsf{inc}^{-1}\ (2\ 1)\ \mathsf{suc!}\ \mathsf{suc!} \\ \quad \mathsf{inc}^{-1} \to \mathsf{cons?}\ 0?\ \mathsf{nil?}\ \mathsf{nil!} \\ \quad \mathsf{inc}^{-1} \to \mathsf{cons?}\ 1?\ 0!\ \mathsf{cons!} \\ \quad \mathsf{inc}^{-1} \to \mathsf{cons?}\ 0?\ \mathsf{inc}^{-1}\ 1!\ \mathsf{cons!} \end{array}\right\}$$

(3): To make unbin2 injective, we use this coding instead of the usual Peano-style coding nil (as 0), cons(1, nil) (as 1), cons(1, cons(0, nil)) (as 2), ...

The determinization of $G_{\mathsf{unbin2}^{-1}}$ fails since (2 1) in the first rule and suc? in the third rule cause a *shift/shift conflict* in constructing *item sets* from $G_{\mathsf{unbin2}^{-1}}$. The operation sequence (2 1) nil? in the first rule is used for examining the form of the second element of the stack but the use of (2 1) conflicts other operations.

In the next section, to represent such operation sequences as a single operation, we extend the syntax and semantics of the matching operation.

3. Extension of Matching Operation

In this section, we extend the syntax and semantics of the matching operation so as to refer to non-topmost elements in the stack.

As described at the end of the previous section, we represent operation sequences $\pi : c?$ such as (2 1) nil? by single operations. To achieve this, we introduce new matching operations $c_{(k)}?$ for $c \in \mathcal{C}$ where $k > 0$, e.g., $\mathsf{nil}_{(2)}?$ for (2 1) nil?. $\mathcal{G}_{\mathsf{ext}}$ denotes the set of grammar programs obtained by adding the following atomic operations to the syntax for $\mathcal{G}$:

$$c_{(k)}? \qquad \text{where } c \in \mathcal{C} \text{ and } k > 0$$

The new operation $c_{(k)}?$ plays a role of the matching operation not only for the topmost element but for the kth topmost element. Thus, the semantics of a program G' in $\mathcal{G}_{\mathsf{ext}}$ is obtained by adding the following case to $\to_G$:

$$\begin{array}{l} (c_{(k)}? : ts, v_1 : \cdots : v_{k-1} : c(u_1, \ldots, u_n) : vs) \\ \qquad \to_{G'} (ts, v_1 : \cdots : v_{k-1} : u_1 : \cdots : u_n : vs) \end{array}$$

where $c/n \in \mathcal{C}$. We denote $\mathcal{A} \cup \{c_{(k)}? \mid c \in \mathcal{C},\ k > 0\}$ by $\mathcal{A}_{\mathsf{ext}}$.

In the rest of this section, we show that the replacement of $(k\ i_i\ \ldots\ i_{k-1}) : c?$ by $c_{(k)}? : (k\ k+1\ \ldots\ k+(n-1)\ i_1\ \ldots\ i_{k-1})$ preserves the equivalence of the semantics.

We first formalize the replacement $\Leftrightarrow$ as the symmetric closure satisfying all of the following:

- $ts : (k\ i_i\ \ldots\ i_{k-1}) : c? : us \Leftrightarrow ts : c_{(k)}? : (k\ k+1\ \ldots\ k+(n-1)\ i_1\ \ldots\ i_{k-1}) : us$ where $c/n \in \mathcal{C}$,$n > 0$, $\{i_1, \ldots, i_{k-1}\} = \{1, \ldots, k-1\}$ and
- $ts : (k\, i_1\ \ldots\ i_{k-1}) : c? : us \Leftrightarrow ts : c_{(k)}? : (i_1\ \ldots\ i_{k-1}) : us$, where $c/0 \in \mathcal{C}$, and $\{i_1, \ldots, i_{k-1}\} = \{1, \ldots, k-1\}$.

For a sequence ts of operations over the extended grammar language, the set $\mathrm{rep}_{\Leftrightarrow}(ts)$ denotes the set of sequences obtained from ts by applying the reflexive and transitive closure of $\Leftrightarrow$: $\mathrm{rep}_{\Leftrightarrow}(ts) = \{ts' \mid ts \Leftrightarrow^* ts'\}$ where continuous sequences of permutations are optimized as much as possible.

Lemma 4. *Let ts be a sequence in $\mathcal{A}^*_{\mathsf{ext}}$, $ts' \in \mathrm{rep}_{\Leftrightarrow}(ts)$, $G = \{f \to ts\}$, and $G' = \{f \to ts'\}$. Then, for all sequences $vs, us \in T(\mathcal{C})^*$, $(ts, vs) \to^*_G (\epsilon, us)$ iff $(ts, vs) \to^*_{G'} (\epsilon, us)$.*

NII-Electronic Library Service

Proof. This lemma is trivial by the definition of the replacement $\Leftrightarrow$. □

Theorem 5. *Let* $G = \{g_1 \to ts_1, \ldots, g_n \to ts_n\}$ *be a grammar program in* $\mathcal{G}$ *that defines an* n*-ary function* f *returning* m *values, and* G' *be a grammar program in* $\mathcal{G}_{\mathsf{ext}}$ *such that* $G' = \{g_i \to ts'_i \mid 1 \leqq i \leqq n,\ ts'_i \in \mathrm{rep}_{\Leftrightarrow}(ts)\}$. *Then, for all terms* $v_1, \ldots, v_n, u_1, \ldots, u_m \in T(\mathcal{C})$, $(f : \epsilon, v_1 : \cdots : v_n : \epsilon) \to^*_G (\epsilon, u_1 : \cdots : u_m : \epsilon)$ *iff* $(f : \epsilon, v_1 : \cdots : v_n : \epsilon) \to^*_{G'} (\epsilon, u_1 : \cdots : u_m : \epsilon)$.

Proof. This theorem can be easily proved by using Lemma 4. □

Example 6. Consider the grammar program $G_{\mathsf{unbin2}^{-1}}$ in Example 3 again. By replacing (2 1) nil? in the first rule by $\mathsf{nil}_{(2)}$?, we obtain the following grammar program:

$$G'_{\mathsf{unbin2}^{-1}} = \{\ \mathsf{unbin2}^{-1} \to \mathsf{ub}^{-1}\ \mathsf{nil}_{(2)}?\quad \cdots\ \}$$

By applying the determinization method to $G'_{\mathsf{unbin2}^{-1}}$ with ignoring the extension of the syntax and semantics, we obtain the following grammar programs without shift/shift conflicts:

$$\left\{\begin{array}{l} \mathsf{unbin2}^{-1} \to \mathsf{zero!\ suc!\ f} \\ \mathsf{f} \to \mathsf{nil}_{(2)}? \\ \mathsf{f} \to \mathsf{suc?\ (2\ 1)\ cons?\ g\ (2\ 1)\ suc!\ suc!\ f} \\ \mathsf{g} \to \mathsf{0?\ h} \\ \mathsf{g} \to \mathsf{1?\ 0!\ cons!} \\ \mathsf{h} \to \mathsf{nil?\ nil!} \\ \mathsf{h} \to \mathsf{cons?\ g\ 1!\ cons!} \end{array}\right\}$$

By replacing $\mathsf{nil}_{(2)}$? by (2 1) nil back and by translating the grammar program back into a constructor TRS, we obtain the following non-overlapping system:

$$\left\{\begin{array}{rl} \mathsf{unbin2}^{-1}(x) & \to \mathsf{f}(\mathsf{suc}(\mathsf{zero}), x) \\ \mathsf{f}(x, \mathsf{nil}) & \to x \\ \mathsf{f}(\mathsf{suc}(x), \mathsf{cons}(y, ys)) & \to \mathsf{f}(\mathsf{suc}(\mathsf{suc}(x)), \mathsf{g}(y, ys)) \\ \mathsf{g}(0, y) & \to \mathsf{h}(y) \\ \mathsf{g}(1, y) & \to \mathsf{cons}(0, y) \\ \mathsf{h}(\mathsf{nil}) & \to \mathsf{nil} \\ \mathsf{h}(\mathsf{cons}(y, ys)) & \to \mathsf{cons}(1, \mathsf{g}(y, ys)) \end{array}\right\}$$

This is desirable as an inverse of unbin2 although we misused the determinization method.

As another solution, it may seem useful to exchange the first and second argument of ub one. Unfortunately, this is not a solution since the similar conflict arises.

In the mentioned above, we apply the transformation $\Leftrightarrow$ of π c? to $\mathsf{c}_{(k)}$? π' by hand in order to avoid the shift/shift conflicts we faced. Since the number of candidates for G' in Theorem 5 is finite, to automate this application, the depth-first search is sufficient to detect sequences to be replaced—such sequences are related to shift/shift conflict branches of item sets. This strategy does not allow us to apply the replacement to grammar programs that has no shift/shift conflict in the collections. This indicates that the extended method in this paper is successful for all the examples for which the original method is successful. Therefore, the extended method is strictly better than the original one because of unbin2.

While we mentioned successful examples only, this extension does not always succeed in the determinization. In the following, we show an example where we succeed in avoiding a shift/shift conflict in the collection but fails to produce a non-overlapping TRS while an overlapping and terminating one is produced.

Example 7. Consider the following TRS, a variant of unbin2:

$$\left\{\begin{array}{rl} \mathsf{unbin3}(x) & \to \mathsf{ub}(x, \mathsf{cons}(0, \mathsf{nil})) \\ \mathsf{ub}(\mathsf{suc}(\mathsf{zero}), y) & \to y \\ \mathsf{ub}(\mathsf{suc}(\mathsf{suc}(x)), y) & \to \mathsf{ub}(\mathsf{suc}(x), \mathsf{inc}(y)) \\ \mathsf{inc}(\mathsf{nil}) & \to \mathsf{cons}(0, \mathsf{nil}) \\ \mathsf{inc}(\mathsf{cons}(0, xs)) & \to \mathsf{cons}(1, xs) \\ \mathsf{inc}(\mathsf{cons}(1, xs)) & \to \mathsf{cons}(0, (\mathsf{inc}(xs))) \end{array}\right\}$$

where $\mathcal{C} = \{\mathsf{nil}/0, \mathsf{cons}/2, 0/0, 1/0, \mathsf{zero}/0, \mathsf{suc}/1, \ldots\}$ and $\mathcal{F} = \{\mathsf{unbin3}/1, \mathsf{ub}/2,\ \mathsf{inc}/1\}$. This program is translated and inverted into the following grammar program:

$$G_{\mathsf{unbin3}^{-1}} = \left\{\begin{array}{l} \mathsf{unbin3}^{-1} \to \mathsf{ub}^{-1}\ (2\ 1)\ \mathsf{cons?\ 0?\ nil?} \\ \mathsf{ub}^{-1} \to \mathsf{zero!\ suc!} \\ \mathsf{ub}^{-1} \to \mathsf{ub}^{-1}\ \mathsf{suc?}\ (2\ 1)\ \mathsf{inc}^{-1}\ (2\ 1)\ \mathsf{suc!\ suc!} \\ \mathsf{inc}^{-1} \to \mathsf{cons?\ 0?\ nil?\ nil!} \\ \mathsf{inc}^{-1} \to \mathsf{cons?\ 1?\ 0!\ cons!} \\ \mathsf{inc}^{-1} \to \mathsf{cons?\ 0?}\ \mathsf{inc}^{-1}\ \mathsf{1!\ cons!} \end{array}\right\}$$

This grammar program corresponds to the following conditional TRS:

$$\left\{\begin{array}{rl} \mathsf{unbin3}^{-1}(x) & \to z \\ & \Leftarrow \mathsf{ub}^{-1}(x) \twoheadrightarrow (z, \mathsf{cons}(0, \mathsf{nil})) \\ \mathsf{ub}^{-1}(x) & \to (\mathsf{suc}(\mathsf{zero}), x) \\ \mathsf{ub}^{-1}(x) & \to (\mathsf{suc}(\mathsf{suc}(y), w) \\ & \Leftarrow \mathsf{ub}^{-1}(x) \twoheadrightarrow (\mathsf{suc}(y), z), \\ & \quad \mathsf{inc}^{-1}(z) \twoheadrightarrow (w) \\ \mathsf{inc}(\mathsf{nil}) & \to \mathsf{cons}(0, \mathsf{nil}) \\ \mathsf{inc}(\mathsf{cons}(0, xs)) & \to \mathsf{cons}(1, xs) \\ \mathsf{inc}(\mathsf{cons}(1, xs)) & \to \mathsf{cons}(0, (\mathsf{inc}(xs))) \end{array}\right\}$$

This conditional TRS is not operationally terminating, either. By replacing (2 1) cons? in the first rule by $\mathsf{cons}_{(2)}$? (2 3 1), we obtain the following grammar program:

$$G'_{\mathsf{unbin3}^{-1}} = \{\ \mathsf{unbin3}^{-1} \to \mathsf{ub}^{-1}\ \mathsf{cons}_{(2)}?\ (2\ 3\ 1)\ 0\ \mathsf{nil?}\quad \cdots\ \}$$

NII-Electronic Library Service

Institute of Electronics, Information, and Communication Engineers

By applying the determinization method to $G_{\mathsf{unbin3}^{-1}}$ again, we obtain the following grammar program:

$$\left\{\begin{array}{l} \mathsf{unbin3}^{-1} \to \mathsf{zero!\ suc!\ f} \\ \quad \mathsf{f} \to \mathsf{cons}_{(2)}?\ (2\ 3\ 1)\ 0\ \mathsf{nil?} \\ \quad \mathsf{f} \to \mathsf{suc?}\ (2\ 1)\ \mathsf{cons?\ g}\ (2\ 1)\ \mathsf{suc!\ suc!\ f} \\ \quad \mathsf{g} \to 0?\ \mathsf{h} \\ \quad \mathsf{g} \to 1?\ 0!\ \mathsf{cons!} \\ \quad \mathsf{h} \to \mathsf{nil?\ nil!} \\ \quad \mathsf{h} \to \mathsf{cons?\ g}\ 1!\ \mathsf{cons!} \end{array}\right\}$$

This grammar program corresponds to the following TRS:

$$\left\{\begin{array}{rl} \mathsf{unbin3}^{-1}(x) & \to \mathsf{f}(\mathsf{suc}(\mathsf{zero}), x) \\ \mathsf{f}(x, \mathsf{cons}(0, \mathsf{nil})) & \to x \\ \mathsf{f}(\mathsf{suc}(x, \mathsf{cons}(y, ys))) & \to \mathsf{f}(\mathsf{suc}(\mathsf{suc}(x)), \mathsf{g}(y, ys)) \\ \mathsf{g}(0, y) & \to \mathsf{h}(y) \\ \mathsf{g}(1, y) & \to \mathsf{cons}(0, y) \\ \mathsf{h}(\mathsf{nil}) & \to \mathsf{nil} \\ \mathsf{h}(\mathsf{cons}(y, ys)) & \to \mathsf{cons}(1, \mathsf{g}(y, ys)) \end{array}\right\}$$

This TRS is still overlapping. The extended method fails to produce a non-overlapping. The reason is that $G_{\mathsf{unbin3}^{-1}}$ potentially contains reduce/reduce conflict between the second rule and the third/ forth rules although the shift/shift conflict mentioned above prevents us from the reduce/reduce conflict. On the other hand, this TRS is terminating. The extended method transforms a non-operationally-terminating conditional TRS to a terminating one. Therefore, the extended determinization is meaningful for $G'_{\mathsf{unbin3}^{-1}}$ in the sense that terminating TRSs are produced.

Our approach is a breakthrough for avoiding shift/shift conflicts but not for producing non-overlapping systems since, e.g., $\mathsf{nil}_{(2)}?$ in Example 6 is a pattern matching for the second topmost element of the stack but suc? is one for the topmost element, i.e., the branches labeled with $\mathsf{nil}_{(2)}?$ and suc? do not correspond to the disjoint pattern matching. Though, the code generation sometimes produces non-overlapping programs, and thus, our approach is sometimes meaningful.

4. Future Work

As matters stand, proving the correctness of the determinization for extended grammar programs is left unanswered. We believe that the application of the determinization method to extended grammar programs is correct if no fresh atomic operators is introduced by means of the application. This is because the correctness of the determinization method for original grammar programs guarantees the equivalence of words generated by the input grammar and its resulting grammar. We will prove the correctness by using this intuitive observation.

We showed two successful examples and an unsuccessful example of the extended LRinv but we have never discussed the reason why the extended method succeeds in producing the non-overlapping TRSs in the two example and why produces the overlapping (and terminating) TRS. As our future work, we will characterize the difference between the successful and unsuccessful examples, clarifying a sufficient condition for producing non-overlapping TRSs.

We should compare the extended method with the existing inversion methods (e.g., [9]). In general, the resulting programs of the inversion methods are different. For this reason, it is very difficult to compare the inversion methods from theoretical point of view. Thus, we will compare the extended method with the other inversion methods by means of the benchmarks shown in several papers on program inversion.

Acknowledgments This work has been partially supported by *MEXT KAKENHI* #21700011.

References

[1] J. M. Almendros-Jiménez and G. Vidal. Automatic partial inversion of inductively sequential functions. In *Proc. of IFL 2006*, vol. 4449 of *LNCS*, pp. 253–270, Springer, 2006.

[2] R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In *Proc. of APLAS 2003*, vol. 2895 of *LNCS*, pp. 246–264, Springer, 2003.

[3] R. Glück and M. Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundam. Inform.*, 66(4):367–395, 2005.

[4] M. Kawabe and R. Glück. The program inverter LRinv and its structure. In *Proc. of PADL 2005*, vol. 3350 of *LNCS*, pp. 219–234, Springer, 2005.

[5] K. Matsuda, S.-C. Mu, Z. Hu, and M. Takeichi. A grammar-based approach to invertible programs. In *Proc. of ESOP 2010*, vol.6012 of *LNCS*, pp. 448–467, Springer, 2010.

[6] N. Nishida and M. Sakai. Completion after program inversion of injective functions. *ENTCS*, 237:39–56, 2009.

[7] N. Nishida, M. Sakai, and T. Sakabe. Generation of inverse computation programs of constructor term rewriting systems. *IEICE Trans. Inf. & Syst.*, J88-D-I(8):1171–1183, 2005 (in Japanese).

[8] N. Nishida, M. Sakai, and T. Sakabe. Partial inversion of constructor term rewriting systems. In *Proc. of RTA 2005*, vol. 3467 of *LNCS*, pp. 264–278, Springer, 2005.

[9] N. Nishida and G. Vidal. Program inversion for tail recursive functions. In *Proc. of RTA 2011*, vol. 10 of *LIPIcs*, pp. 283–298, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[10] E. Ohlebusch. *Advanced topics in term rewriting*. Springer-Verlag, 2002.

[11] A. Romanenko. Inversion and metacomputation. In *Proc. of PEPM 1991*, Sigplan Notices, 26(9), pp. 12–22, ACM, New York, 1991.

[12] A. Romanenko. The generation of inverse functions in Refal. In *Proc. of the Intl. Workshop on Partial Evaluation and Mixed Computation*, pp. 427–444, North-Holland, Amsterdam, 1988.

NII-Electronic Library Service