Derivation of Deterministic Inverse Programs Based on LR Parsing

Robert Glück[1] and Masahiko Kawabe[2]

[1] PRESTO, JST & University of Copenhagen, DIKU
DK-2100 Copenhagen, Denmar
glueck@acm.org

[2] Waseda University, Graduate School of Science and Engineering
Tokyo 169-8555, Japan
kawabe@futamura.info.waseda.ac.jp

Abstract. We present a method for automatic program inversion of functional programs based on methods of LR parsing. We formalize the transformation and illustrate it with the inversion of a program for run-length encoding. We solve one of the main problems of automatic program inversion—the elimination of nondeterminism—by viewing an inverse program as a context-free grammar and applying to it methods of LR parsing to turn it into a recursive, deterministic inverse program. This improves the efficiency of the inverse programs and greatly expands the application range of our earlier method for program inversion.

1 Introduction

The contribution of this paper is an automatic method for program inversion based on methods of LR parsing. This transformation improves the efficiency of the inverse programs and extends the application range of our earlier method by allowing the inversion of programs based on a global transformation of a nondeterministic inverse program. We make use of a self-inverse primitive function for the duplication of values and testing of equality, which we introduced in recent work [8], and a symmetric program representation to simplify inversion. To eliminate nondeterminism in a global view, we apply methods for LR parsing by viewing an inverse program as a context-free grammar and generating a deterministic inverse program if that grammar has certain properties (*e.g.*, without parsing conflicts). This greatly expands the application range of our recent method for program inversion.

The idea of program inversion can be traced back to reference [6]. Recent work [14] has focused on the converse of a function theorem [4], inverse computation of functional programs [2], and the transformation of interpreters into inverse interpreters by partial evaluation [9]. Logic programming is suited to find multiple solutions and can be used for *inverse interpretation*, while in this paper we are interested in *program inversion* (for a detailed description of these notions, see reference [1]). We consider one-to-one functional programs and not relations with multiple solutions. An example is the generation of a program for

$q ::= d_1 \dots d_n$	(program)
$d ::= f(x_1, \dots, x_n) = t$	(definition)
$t ::= (l_1, \dots, l_m)$	(return)
$\quad \mid$ **case** l **of** $\{p_i \to t_i\}_{i=1}^{m}$	(case-expression)
$\quad \mid$ **let** $(y_1, \dots, y_m){=}f(l_1, \dots, l_n)$ **in** t	(let-expression)
$l ::= x$	(variable)
$\quad \mid c(l_1, \dots, l_n)$	(constructor)
$\quad \mid \lfloor l \rfloor$	(duplication/equality)
$p ::= c(x_1, \dots, x_n)$	(pattern)

Fig. 1. Abstract syntax of the source language

decoding data given a program for encoding data, and vice versa. In general, the goal of a program inverter is to find an *inverse program* $q^{-1} : B \to A$ of a program $q : A \to B$ such that for all values $x \in A$ and $y \in B$ we have

$$q(x) = y \iff q^{-1}(y) = x \ .$$

This tells us that, if a program q terminates on input x and returns output y, then the inverse program q^{-1} terminates on y and returns x, and vice versa. This implies that both programs are *injective*; they need not be surjective or total. Here, equality means strong equivalence: either both sides of an equation are defined and equal, or both sides are undefined. In practice, even when it is certain that an efficient inverse program q^{-1} exists, the automatic generation of such a program from q may be difficult or impossible.[3]

The first method developed for automatic program inversion of first-order functional programs appears to be the program inverter by Korf and Eppstein [13,7] (we call it KEinv for short). It is one of only two general-purpose automatic program inverters that have been built (the other one is InvX [12]). Manual methods [6,11,4,14] and semi-automatic methods [5] exist, but require ingenuity and human insight. Our goal is to achieve further automation of general-purpose program inversion.

This paper is organized as follows. First, we define the source language (Sect. 2). Then we discuss our solution of the main challenges of program inversion (Sect. 3) and present our inversion method (Sect. 4). We discuss related work (Sect. 5), and then give a conclusion (Sect. 6). We assume that the reader is familiar with the principles of LR parsing, *e.g.*, as presented in [3].

2 Source Language

We are concerned with a first-order functional language. A program q is a sequence of function definitions d where the body of each definition is a term t

[3] There exists a program inverter that returns, for every q, a *trivial inverse* q^{-1} [1].

```
$pack(s)$ = case $s$ of $[\,] \to ([\,])$
                        $c{:}r \to$ let $(p,l){=}len(c,\mathrm{O},r)$ in $(p{:}pack(l))$
$len(c,n,s)$ = case $s$ of $[\,] \to (\langle c,n\rangle,[\,])$
                          $d{:}r \to$ case $\lfloor\langle c,d\rangle\rfloor$ of
                                  $\langle e\rangle \to$ let $(p,t){=}len(e,\mathrm{S}(n),r)$ in $(p,t)$
                                  $\langle e,f\rangle \to (\langle e,n\rangle,f{:}r)$
```

Fig. 2. Program *pack*

constructed from variables, constructors, function calls, case- and let-expressions (Fig. 1 where $m > 0$, $n \geq 0$). For reasons of symmetry, functions may return multiple output values which is denoted by syntax $(l_1, \ldots, l_m)$. Arity and coarity of functions and constructors are fixed. The language has a call-by-value semantics. A value v in the language is a constructor c with arguments $v_1, ..., v_n$:

$$v ::= c(v_1 \ldots v_n) \ .$$

An example is the program for run-length encoding (Fig. 2): function *pack* encodes a list of symbols as a list of symbol-number pairs, where the number specifies how many copies of a symbol have to be generated upon decoding. For instance, $pack([\mathrm{AABCCC}]) = [\langle \mathrm{A},2\rangle\langle \mathrm{B},1\rangle\langle \mathrm{C},3\rangle]$.[4] Function *pack* maximizes the counter: we never have an encoding like $\langle \mathrm{C},2\rangle\langle \mathrm{C},1\rangle$, but rather, always $\langle \mathrm{C},3\rangle$. This implies that the symbols in two adjacent symbol-number pairs are never equal. Fig. 3 shows the inverse function $pack^{-1}$. In the implementation, we use unary numbers where O denotes One and S the Successor. The primitive function $\lfloor\cdot\rfloor$ checks the equality of two values: $\lfloor\langle v,v'\rangle\rfloor = \langle v\rangle$ if $v = v'$. In the absence of equality, the values are returned unchanged: $\lfloor\langle v,v'\rangle\rfloor = \langle v,v'\rangle$ if $v \neq v'$. This will be defined below.

We consider only *well-formed* programs. As usual, we require that no two patterns p_i and p_j in a case-expression contain the same constructor and that all patterns are linear (no variable occurs more than once in a pattern). We also require that each variable be defined before its use and, for simplicity, that no defined variable be redefined by a case- or let-expression.

Duplication and Equality One of our key observations [8] was that duplication and equality testing are two sides of the same coin in program inversion: the duplication of a value in a program becomes an equality test in the inverse program, and vice versa. To simplify inversion, we introduce a primitive function $\lfloor\cdot\rfloor$ defined as follows:

[4] We use the shorthand notation $x{:}xs$ and $[\ ]$ for the constructors $\mathrm{Cons}(x, xs)$ and Nil. For $x_1{:}x_2{:}\ldots{:}x_n{:}[\ ]$ we write $[x_1x_2\ldots x_n]$, or sometimes $x_1x_2\ldots x_n$. A tuple $\langle x_1,\ldots,x_n\rangle$ is a shorthand notation for an n-ary constructor $\mathrm{C}_n(x_1,\ldots,x_n)$.

```
$f_{[0]}(s) =$ let $(r)$=$f_{[1]}(s)$ in $(r)$
$f_{[1]}(s) =$ case $s$ of $[\,] \rightarrow ([\,])$
                        $p{:}r \rightarrow$ let $(l)$=$f_{[1]}(r)$ in
                                   let $(c,n,a)$=$f_{[26]}(l,p)$ in
                                   let $(b)$=$f_{[11,10,9]}(n,c,a)$ in $(b)$
$f_{[11,10,9]}(n,c,s) =$ case $n$ of $\mathrm{O} \rightarrow (c{:}s)$
                                    $\mathrm{S}(m) \rightarrow$ case $\lfloor\langle c\rangle\rfloor$ of
                                                 $\langle d,e\rangle \rightarrow$ let $(a)$=$f_{[11,10,9]}(m,d,e{:}s)$ in $(a)$
$f_{[26]}(s,p) =$ case $s$ of $[\,] \rightarrow$ case $y$ of $\langle c,n\rangle \rightarrow (c,n,[\,])$
                          $c{:}r \rightarrow$ case $p$ of
                                    $\langle d,n\rangle \rightarrow$ case $\lfloor\langle d,c\rangle\rfloor$ of $\langle e,f\rangle \rightarrow (e,n,f{:}r)$
```

Fig. 3. Program *pack*$^{-1}$

$$\lfloor\langle v\rangle\rfloor \overset{\text{def}}{=} \langle v,v\rangle \qquad\qquad \text{(duplication)}$$

$$\lfloor\langle v,v'\rangle\rfloor \overset{\text{def}}{=} \begin{cases}\langle v\rangle & \text{if } v = v' \\ \langle v,v'\rangle & \text{if } v \neq v'\end{cases} \qquad\qquad \text{(equality test)}$$

There are mainly two ways of using this function: duplication and equality testing. In the former case, given a single value, a pair with identical values is returned; in the latter case, given a pair of identical values, a single value is returned; otherwise the pair is returned unchanged. The advantage of this unusual function definition is that it makes it easier to deal with duplication and equality testing in program inversion. The function has a useful property, namely that it is *self-inverse*, which means it is its own inverse: $\lfloor\cdot\rfloor^{-1} = \lfloor\cdot\rfloor$.

For example, in function *len* (Fig. 2) the equality of two adjacent symbols, c and d, is tested in the innermost case-expression. The assertion that those symbols are not equal is checked in forward and backward computation.

3 Challenges to Program Inversion

The most challenging point in program inversion is the inversion of conditionals (here, case-expressions). To calculate the input from a given output, we must know which of the m branches in a case-expression the source program took to produce that output, since *only one* of the branches was executed in the forward calculation (our language is deterministic). To make this choice in an inverse program, we must know m postconditions, R_i, one for each branch, such that for each pair of postconditions, we have: $R_i \wedge R_j = \mathit{false}$ $(1 \leq i < j \leq m)$. This divides the set of output values into m disjoint sets, and we can choose the correct branch by testing the given output value using the postconditions.

Postconditions that are suitable for program inversion can be derived by hand (*e.g.*, [6,11]). In automatic program inversion they must be inferred from a source program. The program inverter KEinv [7] uses a heuristic method, and the language in which its postconditions are expressed consists of the primitive predicates available for the source language's value domain consisting of lists and integers. In general, there is no automatic method that would always find mutually exclusive postconditions, even if they exist.

A nondeterministic choice is an unspecified choice from a number of alternatives. Not every choice will lead to a successful computation. If there is only one choice to choose from, then the computation is deterministic.

In previous work [8, Sect.4], we gave a local criteria for checking whether the choice in an inverse program will be deterministic. We viewed the body of a function as a tree with the head of the function definition as the root, and required that the expressions in the leaves of the tree return disjoint sets of values. For example, the expressions in the two leaves of function *pack* are ($[\,]$) and ($\_{:}\_$). Clearly, both represent disjoint sets of values. This works surprisingly well for a class of programs, but is too restricted for other cases. For example, consider function *len* (Fig. 2). It has three leaf expressions each of which returns two values (a symbol-number pair and the remaining list of symbols):

$$1.\ (\langle c,n\rangle,[\,])\qquad 2.\ (p,t)\qquad 3.\ (\langle e,n\rangle,f{:}r)$$

Our local criterion can distinguish the set of values returned by leaf (1) and (3), but it is not sufficient for leaf (2). The set of values represented by (2) is not disjoint from (1) and (3). In fact, it is the union of (1) and (3).

This paper deals with this limitation of our previous method by applying methods from LR parsing to determine whether the choice is deterministic and to generate a recursive inverse program. These techniques replace our local criteria. As we shall see, a deterministic inverse program $pack^{-1}$ can be derived from *pack* by the method introduced in this paper.

Dead Variables Another problematic point in program inversion is when input values are discarded. Consider the selection function *first* defined by

$$\mathit{first}(x) = \textbf{case}\ x\ \textbf{of}\ h{:}t \rightarrow h$$

When we invert such a program, we have to guess 'lost values' (here, a value for t). In general, there are infinitely many possible guesses. We adopted a straightforward solution which we call the "preservation of values" requirement. For a program *well-formed for inversion*, we also require that each defined variable be used exactly once. Thus, a variable's value is always part of the output, and the only way to "diminish the amount of output" is to reduce pairs of values into singletons by $\lfloor\cdot\rfloor$. For example, we write $\textbf{case}\ \lfloor\langle x,y\rangle\rfloor\ \textbf{of}\ \langle z\rangle \rightarrow \ldots$. This expression ensures that no information is lost because all values need to be identical.

$q ::= d_1 \dots d_n$	(program)
$d ::= f \rightarrow t_1 \dots t_n$	(definition)
$t ::= \mathbf{in}(x_1, \dots, x_n)$	(input)
$\mid\ \mathbf{out}(y_1, \dots, y_n)$	(output)
$\mid\ c(x_1, \dots, x_n){=}y$	(constructor)
$\mid\ x{=}c(y_1, \dots, y_n)$	(pattern matching)
$\mid\ \lfloor x \rfloor{=}y$	(duplication/equality)
$\mid\ f(x_1, \dots, x_n){=}(y_1, \dots, y_m)$	(function call)

Fig. 4. Abstract syntax of the symmetric language

4 A Method for Program Inversion

We now present a method for the automatic inversion of programs that are well-formed for inversion. Our method uses symmetric representation of a program as internal representation for inverting primitive operators and a grammar representation for eliminating nondeterminism. It consists of translations $\mathit{SYM}[\![\ \cdot\ ]\!]$, $\mathit{GRAM}[\![\ \cdot\ ]\!]$ and $\mathit{FCT}[\![\ \cdot\ ]\!]$ that translate from the source language to the internal representation and vice versa. Local inversion $\mathit{INV}[\![\ \cdot\ ]\!]$ is then performed by backward reading of the symmetric representation. Finally, $\mathit{DET}[\![\ \cdot\ ]\!]$ attempts to eliminate nondeterminism by LR parsing techniques. To simplify inversion of a program, inversion is carried out on a symmetric representation, rather than on the source program. We now give its definition and explain each of its components in the remainder of this section.

Definition 1 (program inverter). *Let q be a program well-formed for inversion. Then* program inverter $[\![\cdot]\!]^{-1}$ *is defined by*

$$[\![q]\!]^{-1} \stackrel{\text{def}}{=} \mathit{FCT}[\![\ \mathit{DET}[\![\ \mathit{GRAM}[\![\ \mathit{INV}[\![\ \mathit{SYM}[\![\ q\ ]\!]\ ]\!]\ ]\!]\ ]\!]\ ]\!]$$

4.1 Translation to the Symmetric Language

The translation of a function to a symmetric representation makes it easier to invert the function. During the translation, each construct is decomposed into a sequence of atomic operations. The syntax of the symmetric language is shown in Fig. 4. An atomic operation t is either a construct that marks several variables as input $\mathbf{in}(x_1, \dots, x_n)$ or as output $\mathbf{out}(y_1, \dots, y_n)$, an equality representing a constructor application $c(x_1, \dots, x_n){=}y$, a pattern matching $x{=}c(y_1, \dots, y_n)$, an operator application $\lfloor x \rfloor{=}y$, or a function call $f(x_1, \dots, x_n){=}(y_1, \dots, y_m)$. As a convention, the left-hand side of an equation is defined only in terms of input variables (here x) and the right-hand side is defined only in terms of output variables (here y). The intended forward reading of a sequence of equalities is from left to right; the backward reading will be from right to left. A function is

$$
\begin{array}{ll}
\mathit{Sym}\llbracket\ f(\mathit{xs}) = t\ \rrbracket & = \mathit{symt}\llbracket\ t,\ \mathbf{in}(\mathit{xs}),\ f\ \rrbracket \\[1em]
\mathit{symt}\llbracket\ (l_1,...,l_n),\ \mathit{ts},\ f\ \rrbracket & = \{f \to \mathit{syml}\llbracket\ l_n,\ \hat{x}_n, ...\ \mathit{syml}\llbracket\ l_1,\ \hat{x}_1,\ \mathit{ts}\ \rrbracket ...\ \rrbracket\ \mathbf{out}(\hat{x}_1,...,\hat{x}_n)\} \\
\mathit{symt}\llbracket\ \mathbf{case}\ l\ \mathbf{of}\ \{p_i \to t_i\}_{i=1}^m,\ \mathit{ts},\ f\ \rrbracket & = \bigcup_{i=1}^{m} \mathit{symt}\llbracket\ t_i,\ \mathit{syml}\llbracket\ l,\ \hat{x},\ \mathit{ts}\ \rrbracket\ \hat{x}{=}p_i,\ f\ \rrbracket \\
\mathit{symt}\llbracket\ \mathbf{let}\ (\mathit{ys}){=}f(\mathit{xs})\ \mathbf{in}\ t,\ \mathit{ts},\ f\ \rrbracket & = \mathit{symt}\llbracket\ t,\ \mathit{ts}\ f(\mathit{xs}){=}(\mathit{ys}),\ f\ \rrbracket \\[1em]
\mathit{syml}\llbracket\ x,\ y,\ \mathit{ts}\ \rrbracket & = \mathit{ts}\{x \mapsto y\} \\
\mathit{syml}\llbracket\ c(l_1,...,l_n),\ y,\ \mathit{ts}\ \rrbracket & = \mathit{syml}\llbracket\ l_n,\ \hat{x}_n,\ ...\mathit{syml}\llbracket\ l_1,\ \hat{x}_1,\ \mathit{ts}\ \rrbracket ...\ \rrbracket\ c(\hat{x}_1,...,\hat{x}_n){=}y \\
\mathit{syml}\llbracket\ \lfloor l \rfloor,\ y,\ \mathit{ts}\ \rrbracket & = \mathit{syml}\llbracket\ l,\ \hat{x},\ \mathit{ts}\ \rrbracket\ \lfloor \hat{x} \rfloor{=}y
\end{array}
$$

Fig. 5. Translation from the functional language to the symmetric language

represented by one or more linear sequences of atomic operations. If a match operation in a sequence fails, the next sequence is tried. For instance, examine the result of translating function *pack* into the symmetric representation in Fig. 12. The translation is defined in Fig. 5. Function $\mathit{symt}\llbracket\ \cdot\ \rrbracket$ performs a recursive decent over t expressions until it reaches a return expression; function $\mathit{syml}\llbracket\ \cdot\ \rrbracket$ translates l expressions. The translation fixes an evaluation order when translating expressions with multiple arguments (other orders are possible). Notation $\hat{x}$ denotes a fresh variable; they act as liaison variables.

Definition 2 (frontend). *Let d be a definition in a program well-formed for inversion. Then, the translation from the functional language to the symmetric language is defined by*

$$\mathit{SYM}\llbracket\ q\ \rrbracket \overset{\text{def}}{=} \bigcup_{d \in q} \mathit{Sym}\llbracket\ d\ \rrbracket$$

4.2 Local Inversion of a Symmetric Program

Operations in the symmetric representation are easily inverted by reading the intended meaning backwards. Every construct in the symmetric language has an inverse construct. Each function definition is inverted separately. The idea of inverting programs by 'backward reading' is not new and can be found in [6,11]. The rules for our symmetric representation are shown in Fig. 6. Global inversion of a program at this stage is based on the local invertibility of atomic operations.

The inverse of $\mathbf{in}(x_1, \ldots, x_n)$ is $\mathbf{out}(x_1, \ldots, x_n)$ and vice versa; the inverse of constructor application $c(x_1, \ldots, x_n){=}y$ is pattern matching $y{=}c(x_1, \ldots, x_n)$ and vice versa; the inverse of function call $f(x_1, \ldots, x_n){=}(y_1, \ldots, y_m)$ is $f^{-1}(y_1, \ldots, y_m){=}(x_1, \ldots, x_n)$. As explained in Sect. 2, primitive function $\lfloor\cdot\rfloor$ is its own inverse. Thus, the inverse of $\lfloor x \rfloor{=}y$ is $\lfloor y \rfloor{=}x$. Observe that the inversion performs no unfold/fold on functions. It terminates on all programs.

$$
\begin{aligned}
&\mathit{Inv}[\![\ f \rightarrow t_1 \ldots t_n\ ]\!] &&= f^{-1} \rightarrow \mathit{inv}[\![\ t_n\ ]\!] \ldots \mathit{inv}[\![\ t_1\ ]\!]\\
&\mathit{inv}[\![\ \mathbf{in}(x_1,\ldots,x_n)\ ]\!] &&= \mathbf{out}(x_1,\ldots,x_n)\\
&\mathit{inv}[\![\ \mathbf{out}(x_1,\ldots,x_n)\ ]\!] &&= \mathbf{in}(x_1,\ldots,x_n)\\
&\mathit{inv}[\![\ c(x_1,\ldots,x_n){=}y\ ]\!] &&= y{=}c(x_1,\ldots,x_n)\\
&\mathit{inv}[\![\ x{=}c(y_1,\ldots,y_n)\ ]\!] &&= c(y_1,\ldots,y_n){=}x\\
&\mathit{inv}[\![\ \lfloor x \rfloor{=}y\ ]\!] &&= \lfloor y \rfloor{=}x\\
&\mathit{inv}[\![\ f(x_1,\ldots,x_n){=}(y_1,\ldots,y_m)\ ]\!] &&= f^{-1}(y_1,\ldots,y_m){=}(x_1,\ldots,x_n)
\end{aligned}
$$

Fig. 6. Rules for local inversion

The result of backward reading the symmetric representation of *pack* is shown in Fig. 12. Compare *pack* before and after the inversion. Each atomic operation is inverted according to the rules in Fig. 6. Program $pack^{-1}$ is inverse to *pack*, but nondeterministic. We cannot translate len^{-1} directly into a functional program since the call $len^{-1}(p,t){=}(c,z,r)$ is not guarded by pattern matching—the reader is welcome to try.

Definition 3 (local inversion). *Let q be a symmetric program well-formed for inversion. Then, local inversion of q is defined by*

$$\mathit{INV}[\![\ q\ ]\!] \overset{\text{def}}{=} \{\mathit{Inv}[\![\ d\ ]\!] \mid d \in q\}$$

4.3 Translation to the Grammar Language

After the inversion of atomic operations, nondeterminism can be eliminated by viewing the program as a grammar. To make it easier to manipulate programs, we hide variables by translating them into a grammar-like language. That language operates on a stack instead of an environment. Each atomic operation in the symmetric language is converted into a sequence of stack operations. The syntax of the grammar language is shown in Fig. 7. A stack operation t is either a constructor application $c\mathbf{!}$, a pattern matching $c\mathbf{?}$, an application of $\lfloor \_ \rfloor$, a function call f, or a selection $(i_1,\ldots,i_n)$. Each stack operation operates on top of the stack for input/output of the corresponding number of values, except for selection which moves each i_jth stack element to the jth position on the stack. This is convenient for reordering the stack. For instance, the sequence $(2)\ \_{:}\_\mathbf{?}$ swaps the two top-most values and, if the new top-most value is a cons, pops it and pushes its head and tail components; otherwise the sequence fails. The result of translating *pack* from the symmetric language into the grammar language is shown in Fig. 12. The translation is defined in Fig. 8 where “ $+\!\!+$ ” appends two lists.

Definition 4 (midend). *Let d be a definition in a program well-formed for inversion. Then, the translation from the symmetric language to the grammar language is defined by*

$$
\begin{array}{rll}
q & ::= d_1 \dots d_n & \text{(program)} \\
d & ::= f \to t_1 \dots t_n & \text{(definition)} \\
t & ::= c\mathbf{!} & \text{(constructor)} \\
 & \mid\; c\mathbf{?} & \text{(pattern matching)} \\
 & \mid\; \lfloor\_\rfloor & \text{(duplication/equality)} \\
 & \mid\; f & \text{(function call)} \\
 & \mid\; (i_1, \dots, i_n) & \text{(selection)}
\end{array}
$$

Fig. 7. Abstract syntax of the grammar language

$$
\begin{array}{ll}
\mathit{Gram}[\![\ f \to \mathbf{in}(xs)\ ts\ ]\!] & = f \to \mathit{gram}[\![\ ts,\ xs\ ]\!] \\
\mathit{gram}[\![\ \mathbf{out}(xs),\ xs\ ]\!] & = \epsilon \\
\mathit{gram}[\![\ c(xs){=}y\ ts,\ zs\ ]\!] & = (is)\ c\mathbf{!}\ \mathit{gram}[\![\ ts,\ y{:}zs|_{xs}\ ]\!] \\
\mathit{gram}[\![\ x{=}c(ys)\ ts,\ zs\ ]\!] & = (i)\ c\mathbf{?}\ \mathit{gram}[\![\ ts,\ ys \mathbin{+\!\!+} zs|_{x}\ ]\!] \\
\mathit{gram}[\![\ \lfloor x\rfloor{=}y\ ts,\ zs\ ]\!] & = (i)\ \lfloor\_\rfloor\ \mathit{gram}[\![\ ts,\ y{:}zs|_{x}\ ]\!] \\
\mathit{gram}[\![\ f(xs){=}(ys)\ ts,\ zs\ ]\!] & = \begin{cases} f\ \mathit{gram}[\![\ ts,\ ys \mathbin{+\!\!+} zs|_{xs}\ ]\!] & \text{if } x_j = z_j \text{ for } 1 \le j \le n \\ (is)\ f\ \mathit{gram}[\![\ ts,\ ys \mathbin{+\!\!+} zs|_{xs}\ ]\!] & \text{otherwise} \end{cases}
\end{array}
$$

Notation: given xs and zs, (is) is an abbreviation for selection $(i_1, \dots, i_n)$ where number i_j is the index of x_j in zs for $1 \le j \le n$; in particular, i is the index of x in zs; notation $zs|_{xs}$ denotes the deletion of all xs in zs.

Fig. 8. Translation from the symmetric language to the grammar language

$$\mathit{GRAM}[\![\ q\ ]\!] \overset{\text{def}}{=} \{\mathit{Gram}[\![\ d\ ]\!] \mid d \in q\}$$

4.4 Eliminating Nondeterminism

An LR(k) parser generator produces a deterministic parser given a context free grammar, provided that the grammar is LR(k). This class of parsing methods is used in practically all parser generators (*e.g.*, yacc) because it allows to parse most programming language grammars. Our goal is to eliminate nondeterminism from an inverse program. For this we will resort to the particular method of LR(0) parsing. This parsing method is simpler than LR(1) parsing in that it does not require the use of a lookahead operation in the generated parsers.

We found that LR(0) covers a large class of inverse programs. For example, a tail-recursive program can be viewed as a right-recursive grammar; the recursive call is at the end. Local inversion of a tail-recursive program always leads to an inverse program that corresponds to a left-recursive grammar, the recursive call is now at the beginning. Immediately, we face the problem of nondeterminism because it represents an unguarded choice between immediately choosing the recursive call or the base case. Such a program cannot be represented in a functional language. This requires a transformation into a functionally equivalent form where each choice is guarded by a conditional (see also Sect. 3). LR(0) parsing allows us to deal directly with this type of grammars and, in many cases, to convert the program into a deterministic version.

This is a main motivation for applying the method of LR(0) parsing, namely to derive deterministic inverse programs. Our method makes use of some of the methods of classical LR(0) parser generation, for example the construction of item sets by a closure operation, but generates a functional program instead of a table- or program-driven parser:

1. Item sets: given the grammar representation of a program, the items sets are computed by a closure operation.
2. Code generation: given conflict-free item sets, a deterministic functional program is generated.

We will now discuss these operations in more detail. We assume that the reader is familiar with the main principles of LR parsing, *e.g.*, as presented in [3]. Due to space limitations we cannot further review LR parsing and use standard terminology without further definitions (*e.g.*, item set, closure operation, shift/reduce action). We show how these operations are adopted to our grammar language.

Remark: In our previous work [8, Sect.4], we applied a local criterion to a source program to ensures that the inverse program corresponds to an LL grammar. Since LL parsing is strictly weaker than LR parsing, we conclude that applying an LR parsing approach to program inversion leads to a strictly stronger inversion algorithm. Recall that LL parsing cannot directly deal with left-recursive grammars and that any LL grammar can be parsed by an LR parser, but not vice versa.

Item Sets We define a parse item of the grammar language (Fig. 7) by

$$f \rightarrow ts_1 \cdot ts_2$$

where '$\cdot$' denotes the current position. To compute the sets of items sets, we define two operations which correspond to determining the parse actions: *shift* $I \overset{t}{\rightsquigarrow} I'$ from item set I to item set I' under symbol t and *reduce* $I \overset{n}{\hookrightarrow} f$ from item set I by function symbol f and number of operations n.

$$I_1 \overset{t}{\rightsquigarrow} I_2 \iff I_2 = \{f \rightarrow ts_1\ t \cdot ts_2 \mid f \rightarrow ts_1 \cdot t\ ts_2 \in \mathit{closure}[\![\ I_1\ ]\!]\} \quad \text{(shift)}$$
$$I \overset{n}{\hookrightarrow} f \iff f \rightarrow t_1 \ldots t_n \cdot \in \mathit{closure}[\![\ I\ ]\!] \quad \text{(reduce)}$$

Given an initial item set I_0, the set $\mathcal{I}$ of all reachable item sets is defined by

$$\mathcal{I} = \{I \mid I_0 \overset{*}{\rightsquigarrow} I\}$$

$$I_0 = \{\mathit{entry} \rightarrow \ \cdot\, \mathit{pack}^{-1}\}$$

$$I_1 = \left\{\begin{array}{l} \mathit{pack}^{-1} \rightarrow (1) \cdot [\,]\mathbf{?}\ ()\ [\,]\mathbf{!}\ (1) \\ \mathit{pack}^{-1} \rightarrow (1) \cdot \_{:}\_\mathbf{?}\ (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1}\ (2)\ \mathrm{O}\mathbf{?}\ (1,2)\ \_{:}\_\mathbf{!}\ (1) \end{array}\right\}$$

$$I_2 = \{\mathit{pack}^{-1} \rightarrow (1)\ [\,]\mathbf{?} \cdot ()\ [\,]\mathbf{!}\ (1)\}$$

$$\ldots$$

$$I_5 = \{\mathit{pack}^{-1} \rightarrow (1)\ [\,]\mathbf{?}\ ()\ [\,]\mathbf{!}\ (1) \cdot\}$$

$$I_6 = \{\mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_\mathbf{?} \cdot (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1}\ (2)\ \mathrm{O}\mathbf{?}\ (1,2)\ \_{:}\_\mathbf{!}\ (1)\}$$

$$\ldots$$

$$I_9 = \{\mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_\mathbf{?}\ (2)\ \mathit{pack}^{-1}\ (2,1) \cdot \mathit{len}^{-1}\ (2)\ \mathrm{O}\mathbf{?}\ (1,2)\ \_{:}\_\mathbf{!}\ (1)\}$$

$$I_{10} = \left\{\begin{array}{l} \mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_\mathbf{?}\ (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1} \cdot (2)\ \mathrm{O}\mathbf{?}\ (1,2)\ \_{:}\_\mathbf{!}\ (1) \\ \mathit{len}^{-1} \rightarrow \mathit{len}^{-1} \cdot (2)\ \mathrm{S}\mathbf{?}\ (2)\ \langle \_ \rangle\mathbf{!}\ (1)\ \lfloor \_ \rfloor (1)\ \langle \_,\_ \rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1) \end{array}\right\}$$

$$I_{11} = \left\{\begin{array}{l} \mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_\mathbf{?}\ (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1}\ (2) \cdot \mathrm{O}\mathbf{?}\ (1,2)\ \_{:}\_\mathbf{!}\ (1) \\ \mathit{len}^{-1} \rightarrow \mathit{len}^{-1}\ (2) \cdot \mathrm{S}\mathbf{?}\ (2)\ \langle \_ \rangle\mathbf{!}\ (1)\ \lfloor \_ \rfloor (1)\ \langle \_,\_ \rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1) \end{array}\right\}$$

$$I_{12} = \{\mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_\mathbf{?}\ (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1}\ (2)\ \mathrm{O}\mathbf{?} \cdot (1,2)\ \_{:}\_\mathbf{!}\ (1)\}$$

$$\ldots$$

$$I_{15} = \{\mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_\mathbf{?}\ (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1}\ (2)\ \mathrm{O}\mathbf{?}\ (1,2)\ \_{:}\_\mathbf{!}\ (1) \cdot\}$$

$$I_{16} = \{\mathit{len}^{-1} \rightarrow \mathit{len}^{-1}\ (2)\ \mathrm{S}\mathbf{?} \cdot (2)\ \langle \_ \rangle\mathbf{!}\ (1)\ \lfloor \_ \rfloor\ (1)\ \langle \_,\_ \rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1)\}$$

$$\ldots$$

$$I_{25} = \{\mathit{len}^{-1} \rightarrow \mathit{len}^{-1}\ (2)\ \mathrm{S}\mathbf{?}\ (2)\ \langle \_ \rangle\mathbf{!}\ (1)\ \lfloor \_ \rfloor\ (1)\ \langle \_,\_ \rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1) \cdot\}$$

$$I_{26} = \left\{\begin{array}{l} \mathit{len}^{-1} \rightarrow (2) \cdot [\,]\mathbf{?}\ (1)\ \langle \_,\_ \rangle\mathbf{?}\ ()\ [\,]\mathbf{!}\ (2,3,1) \\ \mathit{len}^{-1} \rightarrow (2) \cdot \_{:}\_\mathbf{?}\ (3)\ \langle \_,\_ \rangle\mathbf{?}\ (1,3)\ \langle \_,\_ \rangle\mathbf{!}\ (1)\ \lfloor \_ \rfloor\ (1)\ \langle \_,\_ \rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1) \end{array}\right\}$$

$$I_{27} = \{\mathit{len}^{-1} \rightarrow (2)\ [\,]\mathbf{?} \cdot (1)\ \langle \_,\_ \rangle\mathbf{?}\ ()\ [\,]\mathbf{!}\ (2,3,1)\}$$

$$\ldots$$

$$I_{32} = \{\mathit{len}^{-1} \rightarrow (2)\ [\,]\mathbf{?}\ (1)\ \langle \_,\_ \rangle\mathbf{?}\ ()\ [\,]\mathbf{!}\ (2,3,1) \cdot\}$$

$$I_{33} = \{\mathit{len}^{-1} \rightarrow (2)\ \_{:}\_\mathbf{?} \cdot (3)\ \langle \_,\_ \rangle\mathbf{?}\ (1,3)\ \langle \_,\_ \rangle\mathbf{!}\ (1)\ \lfloor \_ \rfloor\ (1)\ \langle \_,\_ \rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1)\}$$

$$\ldots$$

$$I_{44} = \{\mathit{len}^{-1} \rightarrow (2)\ \_{:}\_\mathbf{?}\ (3)\ \langle \_,\_ \rangle\mathbf{?}\ (1,3)\ \langle \_,\_ \rangle\mathbf{!}\ (1)\ \lfloor \_ \rfloor\ (1)\ \langle \_,\_ \rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1) \cdot\}$$

Fig. 9. *pack*$^{-1}$: item sets

where $I_1 \overset{*}{\leadsto} I_2 \iff I_1 = I_2 \vee \exists t\, \exists I' \,.\, I_1 \overset{t}{\leadsto} I' \wedge I' \overset{*}{\leadsto} I_2$. For our running example, several selected item sets are listed in Fig. 9.

As known from LR parsing, some item sets may be *inadequate*, that is, they contain a shift/reduce or a reduce/reduce conflict. In addition to these two classical conflicts, we have a conflict which is specific to our problem domain (a shift/shift conflict): only pattern matching operations are semantically significant *wrt* to the choice of alternatives; while other operations do not contribute to such a choice. Both shift must pass over different matching operations. With Match we denote the set of all matching operations $c\mathbf{?}$ in the grammar language.

$$
\begin{array}{lll}
\mathit{Det}[\![\, I_i,\ \mathit{is} \,]\!] & = \{f_{i:is} \rightarrow a\ \mathit{ctxt}[\![\, I_i,\ I_j,\ \mathit{is} \,]\!] \mid I_i \overset{a}{\rightsquigarrow} I_j\} & \\
\mathit{gen}[\![\, I_i,\ \mathit{is} \,]\!] & = a\ \mathit{ctxt}[\![\, I_i,\ I_j,\ \mathit{is} \,]\!] & \text{if } S_i = \{(a, I_j)\} \\
\mathit{gen}[\![\, I_i,\ \mathit{is} \,]\!] & = f_{i:is} & \text{if } S_i \supset \{(a, I_j)\} \\
\mathit{gen}[\![\, I_i,\ \mathit{is} \,]\!] & = \mathit{cut}[\![\, I_i,\ \mathit{is},\ f,\ n \,]\!] & \text{if } I_i \overset{n}{\hookrightarrow} f \\
\mathit{ctxt}[\![\, I_0,\ I_j,\ \mathit{is} \,]\!] & = \mathit{gen}[\![\, I_j,\ \mathit{is} \,]\!] & \\
\mathit{ctxt}[\![\, I_i,\ I_j,\ \mathit{is} \,]\!] & = \mathit{gen}[\![\, I_j,\ [\,]\,]\!]\ \mathit{cut}[\![\, I_i,\ i{:}\mathit{is},\ f,\ n \,]\!] & \text{if } R_j = \{(f, n)\} \\
\mathit{ctxt}[\![\, I_i,\ I_j,\ \mathit{is} \,]\!] & = \mathit{gen}[\![\, I_j,\ i{:}\mathit{is} \,]\!] & \text{otherwise} \\
\mathit{cut}[\![\, I_i,\ [i_1, ..., i_m],\ f,\ n \,]\!] & = \epsilon & \text{if } m < n \\
\mathit{cut}[\![\, I_i,\ [i_1, ..., i_n, ..., i_m],\ f,\ n \,]\!] & = \mathit{ctxt}[\![\, I_{i_n},\ I_j,\ [i_{n+1}, ..., i_m] \,]\!] & \text{if } I_{i_n} \overset{f}{\rightsquigarrow} I_j \\
\qquad\qquad \text{where} & \quad S_i = \{(a, I_j) \mid I_i \overset{a}{\rightsquigarrow} I_j\} & \\
 & \quad R_i = \{(f, n) \mid f \rightarrow t_1 \ldots t_n \cdot ts \in I_i\} &
\end{array}
$$

Fig. 10. Code generation

$$
\begin{array}{ll}
I \overset{t}{\rightsquigarrow} I' \wedge I \overset{n}{\hookrightarrow} f & \text{(shift/reduce)} \\
I \overset{n_1}{\hookrightarrow} f_1 \wedge I \overset{n_2}{\hookrightarrow} f_2 \wedge (f_1, n_1) \neq (f_2, n_2) & \text{(reduce/reduce)} \\
I \overset{t_1}{\rightsquigarrow} I_1 \wedge I \overset{t_2}{\rightsquigarrow} I_2 \wedge t_1 \neq t_2 \wedge \{t_1, t_2\} \not\subseteq \text{Match} & \text{(shift/shift)}
\end{array}
$$

Code Generation Given the shift and reduce relations, we now define the code generation. Code generation is only applied if all sets of items are conflict-free. Instead of generating a table- or procedure-driven parser, we generate a program in our grammar representation, which will then be converted into a functional program. The main task of the code generation is to produce for each item set a new function definition in the grammar language. The algorithm makes use of the shift and reduce relations for the given grammar program. It compresses redundant transitions between calls on the fly.

Fig. 12 shows the result for our running example. Inversion is successful. Finally, the grammar representation is translated into a syntactically correct functional program. This translation (not defined here) reintroduces variables and converts each operation into functional language construct. It also determines the arity and coarity of functions. This representation is easier to read, but less easy to manipulate. The inverse program pack^{-1} is shown in Fig. 3. We have automatically produced an unpack function from a pack function. For instance, to unpack a packed symbol list: $\mathit{pack}^{-1}([\langle \mathrm{A}, 2\rangle\langle \mathrm{B}, 1\rangle\langle \mathrm{C}, 3\rangle]) = [\mathrm{AABCCC}]$.

For simplicity, we assume that all item sets can be identified by a unique index (I_1, I_2, *etc.*). These indices will be used to generate new function names and tell us about the context of a function call. For each item set I_i we compute a 'Shift' set S_i and a 'Reduce' set R_i. Set S_i tells us the item set I_j to which

$R_1 = \{(pack^{-1}, 1)\}$	$S_1 = \{([]\mathbf{?}, I_2), (\_\!:\!\_\mathbf{?}, I_6)\}$	
$R_2 = \{(pack^{-1}, 2)\}$	$S_2 = \{((), I_3)\}$	
$\dots$		
$R_5 = \{(pack^{-1}, 5)\}$		$I_5 \overset{5}{\hookrightarrow} pack^{-1}$
$R_6 = \{(pack^{-1}, 2)\}$	$S_6 = \{((2), I_7)\}$	
$R_7 = \{(pack^{-1}, 3)\}$	$S_7 = \{((1), I_1)\}$	$I_7 \overset{pack^{-1}}{\rightsquigarrow} I_8$
$\dots$		
$R_9 = \{(pack^{-1}, 5)\}$	$S_9 = \{((2), I_{26})\}$	$I_9 \overset{len^{-1}}{\rightsquigarrow} I_{10}$
$R_{10} = \{(pack^{-1}, 6), (len^{-1}, 1)\}$	$S_{10} = \{((2), I_{11})\}$	
$R_{11} = \{(pack^{-1}, 7), (len^{-1}, 2)\}$	$S_{11} = \{(\mathrm{O}\mathbf{?}, I_{12}), (\mathrm{S}\mathbf{?}, I_{16})\}$	
$R_{12} = \{(pack^{-1}, 8)\}$	$S_{12} = \{((1, 2), I_{13})\}$	
$\dots$		
$R_{15} = \{(pack^{-1}, 11)\}$		$I_{15} \overset{11}{\hookrightarrow} pack^{-1}$
$R_{16} = \{(len^{-1}, 3)\}$	$S_{16} = \{((2), I_{17})\}$	
$\dots$		
$R_{25} = \{(len^{-1}, 12)\}$		$I_{25} \overset{12}{\hookrightarrow} len^{-1}$
$R_{26} = \{(len^{-1}, 1)\}$	$S_{26} = \{([]\mathbf{?}, I_{27}), (\_\!:\!\_\mathbf{?}, I_{33})\}$	
$R_{27} = \{(len^{-1}, 2)\}$	$S_{27} = \{((1), I_{28})\}$	
$\dots$		
$R_{32} = \{(len^{-1}, 7)\}$		$I_{32} \overset{7}{\hookrightarrow} len^{-1}$
$R_{33} = \{(len^{-1}, 2)\}$	$S_{33} = \{((3), I_{34})\}$	
$\dots$		
$R_{44} = \{(len^{-1}, 13)\}$		$I_{44} \overset{13}{\hookrightarrow} len^{-1}$

Fig. 11. $pack^{-1}$: sets for code generation

we reach by performing operation a; set R_i tells us the names f of the functions used in an item set and the number n of operations passed. Functions $gen\llbracket\ \cdot\ \rrbracket$ and $ctxt\llbracket\ \cdot\ \rrbracket$ make use of these sets. They are defined in Fig. 10; the R and S sets for our running example are shown in Fig. 11.

Definition 5 (backend). *Let q be a grammar program and I_0 be the initial item set for q. Then, the generation of a deterministic program for a (possibly) nondeterministic program q is defined by*

$$DET\llbracket\ q\ \rrbracket \overset{\mathrm{def}}{=} DET'\llbracket\ Det\llbracket\ I_0,\ [\,]\ \rrbracket\ \rrbracket$$

$$DET'\llbracket\ q\ \rrbracket \overset{\mathrm{def}}{=} \begin{cases} q & \mathit{if}\ q' = q \\ DET'\llbracket\ q'\ \rrbracket & \mathit{if}\ q' \neq q \end{cases}$$

$$\mathit{where}\ q' = \bigcup_{f \to ts\ \ f'_{i:is}\ \ ts' \in q} Det\llbracket\ I_i,\ is\ \rrbracket\ \ \cup\ q$$

1) Function-to-symmetric translation:

$$\begin{array}{l}
\mathit{pack} \rightarrow \mathbf{in}(s),\ s{=}[\,],\ [\,]{=}x,\ \mathbf{out}(x)\\
\mathit{pack} \rightarrow \mathbf{in}(s),\ s{=}c{:}r,\ \mathrm{O}{=}x,\ \mathit{len}(c,x,r){=}(p,l),\ \mathit{pack}(l){=}(y),\ p{:}y{=}z,\ \mathbf{out}(z)\\
\mathit{len} \rightarrow \mathbf{in}(c,n,s),\ s{=}[\,],\ \langle c,n\rangle{=}x,\ [\,]{=}y,\ \mathbf{out}(x,y)\\
\mathit{len} \rightarrow\ \mathbf{in}(c,n,s),\ s{=}d{:}r,\ \langle c,d\rangle{=}x,\ \lfloor x\rfloor{=}y,\\
\qquad\quad y{=}\langle c\rangle,\ \mathrm{S}(n){=}z,\ \mathit{len}(c,z,r){=}(p,t),\ \mathbf{out}(p,t)\\
\mathit{len} \rightarrow\ \mathbf{in}(c,n,s),\ s{=}d{:}r,\ \langle c,d\rangle{=}x,\ \lfloor x\rfloor{=}y,\\
\qquad\quad y{=}\langle e,f\rangle,\ \langle e,n\rangle{=}z,\ f{:}r{=}w,\ \mathbf{out}(z,w)
\end{array}$$

2) Local inversion:

$$\begin{array}{l}
\mathit{pack}^{-1} \rightarrow \mathbf{in}(x),\ x{=}[\,],\ [\,]{=}s,\ \mathbf{out}(s)\\
\mathit{pack}^{-1} \rightarrow \mathbf{in}(z),\ z{=}p{:}y,\ \mathit{pack}^{-1}(y){=}(l),\ \mathit{len}^{-1}(p,l){=}(c,x,r),\ x{=}\mathrm{O},\ c{:}r{=}s,\ \mathbf{out}(s)\\
\mathit{len}^{-1} \rightarrow \mathbf{in}(x,y),\ y{=}[\,],\ x{=}\langle c,n\rangle,\ [\,]{=}s,\ \mathbf{out}(c,n,s)\\
\mathit{len}^{-1} \rightarrow\ \mathbf{in}(p,t),\ \mathit{len}^{-1}(p,t){=}(c,z,r),\ z{=}\mathrm{S}(n),\ \langle c\rangle{=}y,\\
\qquad\quad \lfloor y\rfloor{=}x,\ x{=}\langle c,d\rangle,\ d{:}r{=}s,\ \mathbf{out}(c,n,s)\\
\mathit{len}^{-1} \rightarrow\ \mathbf{in}(z,w),\ w{=}f{:}r,\ z{=}\langle e,n\rangle,\ \langle e,f\rangle{=}y\\
\qquad\quad \lfloor y\rfloor{=}x,\ x{=}\langle c,d\rangle,\ d{:}r{=}s,\ \mathbf{out}(c,n,s)
\end{array}$$

3) Symmetric-to-grammar translation:

$$\begin{array}{l}
\mathit{pack}^{-1} \rightarrow (1)\ [\,]\mathbf{?}\ ()\ [\,]\mathbf{!}\ (1)\\
\mathit{pack}^{-1} \rightarrow (1)\ \_{:}\_\mathbf{?}\ (2)\ \mathit{pack}^{-1}\ (2,1)\ \mathit{len}^{-1}\ (2)\ \mathrm{O}\mathbf{?}\ (1,2)\ \_{:}\_\mathbf{!}\ (1)\\
\mathit{len}^{-1} \rightarrow (2)\ [\,]\mathbf{?}\ (1)\ \langle\_,\_\rangle\mathbf{?}\ ()\ [\,]\mathbf{!}\ (2,3,1)\\
\mathit{len}^{-1} \rightarrow \mathit{len}^{-1}\ (2)\ \mathrm{S}\mathbf{?}\ (2)\ \langle\_\rangle\mathbf{!}\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1)\\
\mathit{len}^{-1} \rightarrow (2)\ \_{:}\_\mathbf{?}\ (3)\ \langle\_,\_\rangle\mathbf{?}\ (1,3)\ \langle\_,\_\rangle\mathbf{!}\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1)
\end{array}$$

4) Elimination of non-determinism:

$$\begin{array}{ll}
f_{[0]} & \rightarrow (1)\ f_{[1]}\\
f_{[1]} & \rightarrow [\,]\mathbf{?}\ ()\ [\,]\mathbf{!}\ (1)\\
f_{[1]} & \rightarrow \_{:}\_\mathbf{?}\ (2)\ (1)\ f_{[1]}\ (2,1)\ (2)\ f_{[26]}\ (2)\ f_{[11,10,9]}\\
f_{[11,10,9]} & \rightarrow \mathrm{O}\mathbf{?}\ (1,2)\ \_{:}\_\mathbf{!}\ (1)\\
f_{[11,10,9]} & \rightarrow \mathrm{S}\mathbf{?}\ (2)\ \langle\_\rangle\mathbf{!}\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1)\ (2)\ f_{[11,10,9]}\\
f_{[26]} & \rightarrow [\,]\mathbf{?}\ (1)\ \langle\_,\_\rangle\mathbf{?}\ ()\ [\,]\mathbf{!}\ (2,3,1)\\
f_{[26]} & \rightarrow \_{:}\_\mathbf{?}\ (3)\ \langle\_,\_\rangle\mathbf{?}\ (1,3)\ \langle\_,\_\rangle\mathbf{!}\ (1)\ \lfloor\_\rfloor\ (1)\ \langle\_,\_\rangle\mathbf{?}\ (2,4)\ \_{:}\_\mathbf{!}\ (2,3,1)
\end{array}$$

Fig. 12. Inversion of program *pack*

The transformation into a deterministic grammar by $DET[\![\cdot]\!]$ does not terminate iff there exists a loop, $I_j \overset{+}{\leadsto} I_j$, such that all sets R_k of I_k in this loop contain two or more elements. With another transformation that introduces some administrative code, even these programs for which our algorithm does not terminate can be converted into a deterministic grammar program. Our goal was to avoid the introduction of administrative overhead and we found that our transformation is successful for many programs. We omit the definition of $FCT[\![\cdot]\!]$ which translates a grammar program back into a functional program.

5 Related Work

The method presented in this paper is based on the principle of global inversion based on local invertibility [6,11]. The work was originally inspired by KEinv [13,7]. In contrast to KEinv, our method can successfully deal with equality and duplication of variables. Most studies on functional languages and program inversion have involved program inversion by hand (*e.g.*, [14]). They may be more powerful at the price of automation. This is the usual trade-off. Inversion based on Refal graphs [16,10,17,15] is related to the present method in that both use atomic operations for inversion. An algorithm for inverse computation can be found in [1,2]. It performs inverse computation also on programs that are not injective; it does not produce inverse programs but performs the inversion of a program interpretively.

6 Conclusion

We presented an automatic method for deriving deterministic inverse programs by adopting techniques known from LR parsing, in particular, LR(0) parsing. We formalized the transformation and illustrated it with an example. This greatly expands the application of our recent method for program inversion [8] by eliminating nondeterminism from inverse programs by a global transformation. This allows us to invert programs for which this was not possible before. For example, the method in this paper can invert function *tailcons* and the tail-recursive version of function *reverse* [8, Sect.6].

We have also reached the border line where more inverse programs can be made deterministic, but for the price of introducing additional administrative overhead or the use of LR(k), $k > 0$, that is, parsing methods that involve lookahead operations. It will be a task for future work to study the relative gains by adopting such techniques. We used a grammar-like program representation. Other representations are possible and future work will need to identify which representation is most suitable for eliminating nondeterminism.

Acknowledgements We are grateful to the anonymous reviewers for their detailed and useful feedback.

References

1. S. M. Abramov, R. Glück. Principles of inverse computation and the universal resolving algorithm. In T. Æ. Mogensen, D. Schmidt, I. H. Sudborough (eds.), *The Essence of Computation: Complexity, Analysis, Transformation*, LNCS 2566, 269–295. Springer-Verlag, 2002.
2. S. M. Abramov, R. Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193–229, 2002.
3. A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.
4. R. Bird, O. de Moor. *Algebra of Programming.* Prentice Hall International Series in Computer Science. Prentice Hall, 1997.
5. J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(1):1–46, 1981.
6. E. W. Dijkstra. Program inversion. In F. L. Bauer, M. Broy (eds.), *Program Construction: International Summer School*, LNCS 69, 54–57. Springer-Verlag, 1978.
7. D. Eppstein. A heuristic approach to program inversion. In *Int. Joint Conference on Artificial Intelligence (IJCAI-85)*, 219–221. Morgan Kaufmann, Inc., 1985.
8. R. Glück, M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori (ed.), *Programming Languages and Systems. Proceedings*, LNCS 2895, 246–264. Springer-Verlag, 2003.
9. R. Glück, Y. Kawada, T. Hashimoto. Transforming interpreters into inverse interpreters by partial evaluation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 10–19. ACM Press, 2003.
10. R. Glück, V. F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the Int. Symposium on Symbolic and Algebraic Computation (ISSAC'90)*, 286–287. ACM Press, 1990.
11. D. Gries. *The Science of Programming*, chapter 21 Inverting Programs, 265–274. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
12. H. Khoshnevisan, K. M. Sephton. InvX: An automatic function inverter. In N. Dershowitz (ed.), *Rewriting Techniques and Applications. Proceedings*, LNCS 355, 564–568. Springer-Verlag, 1989.
13. R. E. Korf. Inversion of applicative programs. In *Int. Joint Conference on Artificial Intelligence (IJCAI-81)*, 1007–1009. William Kaufmann, Inc., 1981.
14. S.-C. Mu, R. Bird. Inverting functions as folds. In E. A. Boiten, B. Möller (eds.), *Mathematics of Program Construction. Proceedings*, LNCS 2386, 209–232. Springer-Verlag, 2002.
15. A. P. Nemytykh, V. A. Pinchuk. Program transformation with metasystem transitions: experiments with a supercompiler. In D. Bjørner, M. Broy, I. V. Pottosin (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 1181, 249–260. Springer-Verlag, 1996.
16. A. Y. Romanenko. Inversion and metacomputation. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 12–22. ACM Press, 1991.
17. V. F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.