A Grammar-Based Approach to Invertible Programs

Kazutaka Matsuda[1,2], Shin-Cheng Mu[3,1], Zhenjiang Hu[4], and Masato Takeichi[1]

[1] University of Tokyo, Japan
[2] JSPS Research Fellow
[3] Academia Sinica, Taiwan
[4] National Institute of Informatics, Japan

Abstract. Program inversion has many applications such as in the implementation of serialization/deserialization and in providing support for redo/undo, and has been studied by many researchers. However, little attention has been paid to two problems: how to characterize programs that are easy or hard to invert and whether, for each class of programs, efficient inverses can be obtained. In this paper, we propose an inversion framework that we call *grammar-based inversion*, where a program is associated with an unambiguous grammar describing the range of the program. The complexity of the grammar indicates how hard it is to invert the program, while the complexity is related to how efficient an inverse can be obtained.

1 Introduction

The problem of *program inversion* — deriving a program computing f^{-1} from a program computing f, has been studied over decades [1,7,8,11,15–17,25,27,30] and has many applications including providing support for undo/redo, deriving a deserializing program from a serializing program or vice versa, and serving as an auxiliary phase in other program transformations, such as bidirectionalization [21].

Every method of program inversion faces two challenges: how to handle a wide class of programs, and how to derive efficient inverses for them. Although it is possible to invert all the programs based on symbolic computation with search (e.g., [1]) as in logic programs, an inverse obtained this way could perform much worse than a handwritten inverse. Thus, an inversion method should restrict itself to a certain subclass of programs for which efficient inverses can be derived. It is certainly desirable for an inverter to handle a wider class of programs. Although often overlooked, it is also desirable for the criteria under which a program can be inverted by a particular inverter to be perspicuously specified. This is especially important when program inversion is used by other program transformations, and we have to convert the program into a form acceptable by the inverter.

Two questions arise naturally: For what kind of programs, how efficient inverse programs can be obtained, and how difficult is the inversion process? Those who have worked on program inversion would agree that among the following programs, *double* is the easiest to invert, followed by *snoc*, and *reverse* is the most difficult of the three.

$$\begin{array}{l} \mathit{double}(x) = \textbf{case}\ x\ \textbf{of} \\ \quad \mathsf{Z} \quad\ \ \to \mathsf{Z} \\ \quad \mathsf{S}(y) \to \mathsf{S}(\mathsf{S}(\mathit{double}(y))), \end{array} \qquad \begin{array}{l} \mathit{snoc}(x, b) = \textbf{case}\ x\ \textbf{of} \\ \quad [\,] \quad\ \to [b] \\ \quad a : y \to a : \mathit{snoc}(y, b), \end{array} \qquad \begin{array}{l} \mathit{reverse}(x) = \mathit{rev}(x, [\,]) \\ \mathit{rev}(x, r) = \textbf{case}\ x\ \textbf{of} \\ \quad [\,] \quad\ \to r \\ \quad a : y \to \mathit{rev}(y, a : r). \end{array}$$

A particular method may be able to handle some of these while it may fail on the others. It has not been clarified, however, whether this is merely due to the inadequacy of the method, or whether some problems are intrinsically hard. To the best of the authors' knowledge, there have been no formal classifications of invertible programs so far.

We propose a framework toward solving the classification problem, that we call *grammar-based inversion* in this paper, which is an adaptation of Yellin's inversion [30] for first-order functional programs. Our inversion is based on the correspondence between two proofs: a proof of $\exists x.\ f(x) = v$ for function f, and a proof of $v \in \mathsf{Range}(f)$ where $\mathsf{Range}(f)$ is described by a grammar. More concretely, as in Fig. 1, our inversion uses bijection between a proof for evaluation of a program (an evaluation tree) and a proof for production of a grammar (a production tree). From an output of the program (function), a production tree is obtained by parsing with the grammar. According to the correspondence, the production tree is then converted to an evaluation tree of the program. We also reconstruct the environment used in the evaluation with the evaluation tree, from which we recover the arguments to which the function was applied. The class of programs that can be inverted by the proposed approach is characterized by the complexity of grammars, as seen in Fig. 2. For example, to invert *double* and *snoc*, regular tree grammars (RTG) [6] is sufficient. To invert *reverse*, however, we need grammar beyond regular, such as (inside-out) context-free tree grammar [9]. While a more general grammar covers more programs, it also implies higher worst-case time complexity of parsing and, therefore, a less-efficient inverse. Grammar-based inversion has three main characteristics:

– A program is associated with a grammar, whose complexity characterizes how difficult it is to invert the program.
– The derived inverse is efficiently evaluated by parsing the output with respect to the grammar.
– The correctness of the inversion is clearly expressed by bijection between two proofs.

We present grammar-based inversion with RTG as a case study in this paper. Invertible programs of grammar-based inversion using RTG cover *double* and *snoc* but not *reverse*. However, it will be explained in Sect. 5 that grammar-based inversion, being an extensible framework, can handle functions like *reverse*.

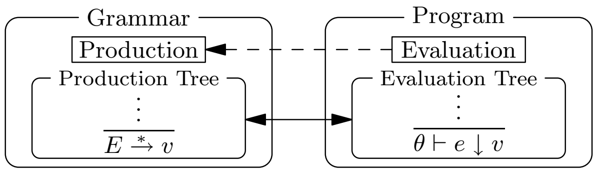
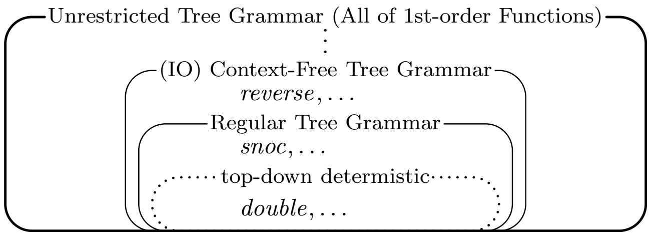


Fig. 1. Idea underlying our inversion: Inversion problem can be rephrased as "given expression e and value v, find environment θ under which e evaluates to v" (Sect. 2).



Unrestricted Tree Grammar (All of 1st-order Functions)
(IO) Context-Free Tree Grammar
reverse, ...
Regular Tree Grammar
snoc, ...
top-down determistic
double, ...

Fig. 2. Classification includes *double*, *snoc*, and *reverse*.

The main purpose of our work is *not* to invert as many programs as possible, or to obtain the most efficient inverses for certain programs. Instead, we aim at classifying programs by grammars that determine the worst-case time complexity of the derived inverses. Also note that we classify programs, not problems. That is, classification by grammar-based inversion is rather syntactic, not semantic.

This paper is organized as follows. Section 2 defines a small core language that we base our discussion on. Section 3 explains an informal account of grammar-based inversion using a small example. Section 4 presents a case study of grammar-based inversion using RTG in detail. Section 5 discusses grammar-based inversion in general. Section 6 describes an experiment that demonstrated the inverses obtained with our method are sufficiently efficient. Section 7 discusses related work. Section 8 concludes the paper and discusses some future directions.

Preliminaries. For function f, function g is called a *left inverse* of f if and only if $g(f(x)) = x$ for all x in the domain of f. For function f, function g is a *right inverse* of f if and only if $f(g(y)) = y$ for all y in the range of f. Both inverses are discussed in this paper. We only consider left/right inverses that are defined precisely on the range of f. Thus, the left inverse is always unique for injective function f. Unless otherwise noted, "inverse" and "invertible" in this paper refer to left inverses.

2 Core Language

To begin with, let us define a small core language to describe the programs to be inverted. The language is merely a first-order functional programming language with call-by-value semantics except for its slightly unusual evaluation rules.

$$\begin{array}{lll} \mathit{prog} & ::= & \mathit{decl}_1 \dots \mathit{decl}_n \\ \mathit{decl} & ::= & f(x_1, \dots, x_n) = e \\ e & ::= & x \mid \mathsf{C}(e_1, \dots, e_n) \mid f(e_1, \dots, e_n) \mid \textbf{case}\ x\ \textbf{of}\ \{p_1 \to e_1; \dots; p_n \to e_n\} \\ p & ::= & x \mid \mathsf{C}(p_1, \dots, p_n) \end{array}$$

Fig. 3. Syntax of core language.

Let Σ be a set of constructors each associated with an arity. The set of *values* are trees $\mathcal{T}_\Sigma$, inductively defined by: Let $\mathsf{C} \in \Sigma$ be an n-ary constructor, $\mathsf{C}(t_1, \dots, t_n) \in \mathcal{T}_\Sigma$ if $t_1, \dots, t_n \in \mathcal{T}_\Sigma$. Note that the definition implies that $\mathsf{C}()$ for nullary C is always in $\mathcal{T}_\Sigma$. For example, given an appropriate Σ, $\mathsf{Z}()$ and $\mathsf{S}(\mathsf{Z}())$ are both trees. For brevity, tree $\mathsf{C}()$ is written as C, and trees $\mathsf{Cons}(x, y)$ and Nil are written as $x : y$ and $[\,]$, respectively. In the later discussion, we assume set Σ containing all constructors in the examples.

A *program* is a set of definitions of first-order functions that take a tuple of values and return a value. The syntax of the language is formally described in Fig. 3. To simplify the presentation, the language does not have a **let** construct, and **case** always matches a variable against patterns. The restrictions do not affect the expressiveness of the language. The set of free variables in expression e is denoted by $\mathsf{vars}(e)$. For simplicity, we assume that the variables in p of **case**-alternative $p \to e$ are always fresh.

We call a program *nonerasing* if every variable in the LHS of a declaration also occurs in the corresponding RHS, and every variable in pattern p of **case**-alternative $p \to e$ occurs in e. If no variable in a program occurs more than once in the RHS, we call the program *affine*.

Substitution θ is a mapping from a finite domain of variables to values. Given pattern p, the value obtained by substituting variables in the domain of θ for corresponding values is denoted by $p\theta$. For set of variables X and substitution θ, domain restriction operator $-|_{\_}$ is defined by $\theta|_X = \{x \mapsto \theta(x) \mid x \in X\}$. Partial operator $\uplus$ merges two substitutions if their domains are disjoint.

The semantics of the language is defined by the big-step call-by-value semantics given in Fig. 4. The semantics is rather standard, except that we eagerly remove unused variables in the environment by domain restriction, which will come in handy in our inversion later. To evaluate expression e, the rules in Fig. 4 are repeatedly applied and an *evaluation tree* (a derivation tree/a proof tree) is constructed. Evaluation tree $\mathcal{E}$ can be seen as a proof that e evaluates to some v under environment θ, which we denote by $\mathcal{E} : \theta \vdash e \Downarrow v$. For simplicity, patterns in **case** are assumed to be non-overlapping, i.e., there is at most one pattern that matches any given input. Note that, given e and θ, evaluation tree $\mathcal{E} : \theta \vdash e \Downarrow v$ is unique if it exists.

3 Grammar-Based Inversion: An Overview

Before going into details, we briefly overview grammar-based inversion.

$$\textsc{Var}: \frac{\theta(x) = v}{\theta \vdash x \downarrow v} \qquad \textsc{Con}: \frac{\{\theta|_{\mathsf{vars}(e_i)} \vdash e_i \downarrow v_i\}_{i \in \{1,\ldots,n\}}}{\theta \vdash \mathsf{C}(e_1,\ldots,e_n) \downarrow \mathsf{C}(v_1,\ldots,v_n)}$$

$$\textsc{Fun}: \frac{\begin{array}{c}\{\theta|_{\mathsf{vars}(e_i)} \vdash e_i \downarrow v_i\}_{i \in \{1,\ldots,n\}} \\ \{x_i \mapsto v_i \mid 1 \leq i \leq n\}|_{\mathsf{vars}(e')} \vdash e' \downarrow v\end{array}}{\theta \vdash f(e_1,\ldots,e_n) \downarrow v} \; (\exists f(x_1,\ldots,x_n) = e')$$

$$\textsc{Case}: \frac{\exists \sigma, i.\ p_i\sigma = \theta(x) \quad (\theta \uplus \sigma)|_{\mathsf{vars}(e_i)} \vdash e_i \downarrow v}{\theta \vdash \textbf{case}\ x\ \textbf{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} \downarrow v}$$

Fig. 4. Big-step call-by-value semantics of core language.

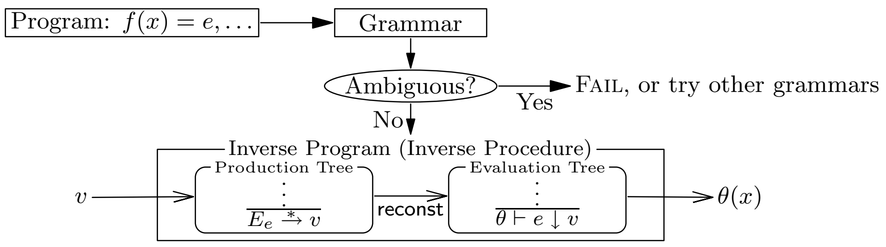

Program: $f(x) = e, \ldots$
Grammar
Ambiguous?
Yes
Fail, or try other grammars
No
Inverse Program (Inverse Procedure)
v
Production Tree
$E_e \xrightarrow{*} v$
reconst
Evaluation Tree
$\theta \vdash e \downarrow v$
$\theta(x)$

Fig. 5. An overview of grammar-based inversion.

3.1 Basic Idea Underlying Grammar-Based Inversion

Recall that, in Sect. 2, a program defines the semantics of expressions. Therefore, we can reduce program inversion to expression inversion as follows.

Problem (Expression Inversion). Given expression e in a program and value v, find environment (substitution) θ such that $\theta \vdash e \downarrow v$.

Given function f, it is reasonable to expect that any notion of a "correct" inversion should cover the entire range of f. That is, it should be complete in the sense that for all v, if there exists θ such that $\theta \vdash f(x) \downarrow v$, we are able to recover θ. This is apparently hard and inefficient for general f. Thus, we restrict ourselves to a method that is only complete for a chosen class of programs.

The goal of grammar-based inversion is to reconstruct the evaluation tree of $\theta \vdash e \downarrow v$, given e and v. This is as hard as only constructing the environment θ. Reconstruction is carried out in two steps: we first construct an approximation of evaluation by building a production tree with respect to a grammar induced by the program, then attempt to reconstruct environment θ from the production tree. Note that the grammar also approximates the range of the program. There is an overview of grammar-based inversion in Fig. 5.

3.2 Inverting *snoc* Step by Step

We demonstrate grammar-based inversion with the example of *snoc* in Sect. 1. The first step is to construct an unambiguous grammar whose production tree approximates an evaluation tree of *snoc*. One would prefer to construct a grammar belonging to a lower complexity class because the complexity is related to the efficiency of the derived inverse. It suffices to use a regular tree grammar (RTG) [6] for *snoc*. From a program, we derive an RTG such that:

- Each nonterminal E_e corresponds to expression e in the program.
- If expression e evaluates to e', we add production rule $E_e \to E_{e'}$.
- Constructors are converted to terminal symbols denoting themselves.
- Nonterminal E_x for variable-use expression x has production rule $E_x \to \top$, where $\top$ is a special symbol that will be explained later.

The conversion will be formalized in the next section. For example, *snoc* is converted to the (unambiguous) grammar below. Here, $\textbf{case}\ x\ \textbf{of}\ \{\ldots\}$ is an abbreviation for $\textbf{case}\ x\ \textbf{of}\ \{[\,] \to b : [\,];\ a : y \to a : \mathit{snoc}(y, b)\}$, the unique RHS of *snoc*, the function we intend to invert.

$$\begin{array}{lll} E_{\textbf{case}\,x\,\textbf{of}\,\{\ldots\}} \to E_{b:[\,]} & E_{a:\mathit{snoc}(y,b)} \to E_a : E_{\mathit{snoc}(y,b)} & E_b \to \top \\ E_{\textbf{case}\,x\,\textbf{of}\,\{\ldots\}} \to E_{a:\mathit{snoc}(y,b)} & E_{\mathit{snoc}(y,b)} \to E_{\textbf{case}\,x\,\textbf{of}\,\{\ldots\}} & E_{[\,]} \to [\,] \\ E_{b:[\,]} \to E_b : E_{[\,]} & E_a \to \top & E_y \to \top \end{array}$$

For the second step, given an output supposedly produced by *snoc*, we first try to parse it against the grammar, allowing $\top$ to match any value. For example, the production tree of $1 : [\,]$, or the derivation/proof tree of production $E_{\textbf{case}\,x\,\textbf{of}\,\{\ldots\}} \xrightarrow{*} 1 : [\,]$, is:

$$\dfrac{\dfrac{\dfrac{\overline{\top \xrightarrow{*} 1}}{E_b \xrightarrow{*} 1} \quad \overline{E_{[\,]} \xrightarrow{*} [\,]}}{E_{b:[\,]} \xrightarrow{*} 1 : [\,]}}{E_{\textbf{case}\,x\,\textbf{of}\,\{\ldots\}} \xrightarrow{*} 1 : [\,]} \;.$$

From the production tree, we can reconstruct the following evaluation tree of $\theta \vdash \mathit{snoc}(x, b) \downarrow 1 : [\,]$:

$$\dfrac{\dfrac{\dfrac{\{b \mapsto 1\}(b) = 1}{\{b \mapsto 1\} \vdash b \downarrow 1} \quad \overline{\emptyset \vdash [\,] \downarrow [\,]}}{\{b \mapsto 1\} \vdash [\,] \to b : [\,] \downarrow 1 : [\,]} \quad \{b \mapsto 1, x \mapsto [\,]\}(x) = [\,]}{\{b \mapsto 1, x \mapsto [\,]\} \vdash \textbf{case}\ x\ \textbf{of}\ \{[\,] \to b : [\,];\ a : y \to a : \mathit{snoc}(y, b)\} \downarrow 1 : [\,]} \;.$$

Each node of the obtained evaluation tree corresponds to a node in the production tree. $\top$ matches 1 in the topmost leaf of the production tree; therefore, the value of b is known to be 1 in the corresponding node in the evaluation tree. By completing the evaluation tree we have also recovered the initial environment, $\{b \mapsto 1, x \mapsto [\,]\}$, the result of inversion.

This reconstruction of evaluation trees from production trees is done by function reconst in Sect. 4. With reconst, our generated inverse program snoc^{-1} has the form:

$$\begin{array}{l} \mathit{snoc}^{-1}(v) = (\theta(x), \theta(b)) \\ \quad \textbf{where}\ (\mathcal{E} : \theta \vdash \mathit{snoc}(x, b) \downarrow v) = \mathsf{reconst}(\mathcal{P}) \\ \qquad\qquad \mathcal{P} \text{ is a production tree for } E_{\textbf{case}\,x\,\textbf{of}\,\{\ldots\}} \xrightarrow{*} v. \end{array}$$

$$\textsc{Any}: \frac{}{\top \overset{*}{\rightharpoonup} v} \qquad \textsc{Unit}: \frac{E_1 \overset{*}{\rightharpoonup} v}{E \overset{*}{\rightharpoonup} v}\ (E \to E_1 \in R)$$

$$\textsc{Con}: \frac{\{E_i \overset{*}{\rightharpoonup} v_i\}_{1 \le i \le n}}{E \overset{*}{\rightharpoonup} \mathsf{C}(v_1, \ldots, v_n)}\ (E \to \mathsf{C}(E_1, \ldots, E_n) \in R)$$

Fig. 6. Semantics of RTG (Σ, N, R).

Since the cost of parsing for RTGs is linear, the derived $snoc^{-1}$ runs in time that is linear to its input. It might seem that $snoc^{-1}$ entails a large overhead. The experiment discussed in Sect. 6 demonstrates that the overhead is acceptable.

4 Grammar-based Inversion by Regular Tree Grammar

This section describes a case study of grammar-based inversion when we use RTG [6], one of the simplest tree grammars, which is relatively well understood; e.g., parsing for RTG can be efficiently performed done tree automaton and its variations [4, 6, 13, 24].

Definition 1 (RTG). *An* RTG *is a triple* (Σ, N, R)*, where* Σ *is a set of constructors (terminals),* N *is a set of nonterminals, and* R *is a set of production rules in which each rule has either of the forms* $E \to E_1$ *or* $E \to \mathsf{C}(E_1, \ldots, E_n)$ *with* $E, E_1, \ldots, E_n \in N$ *and* $\mathsf{C} \in \Sigma$ *of arity* n*.*

Unlike conventional presentation, we define the semantics of a grammar in a "big-step" style as seen in Fig. 6. The rules in Fig. 6 are defined so that $E \overset{*}{\rightharpoonup} v$ means that value v is a normal form of E taking production rules to be rewriting rules. We assume that there exists special nonterminal $\top$ that can generate any value, and we treat $\top \overset{*}{\rightharpoonup} v$ as an axiom. Also note the above definition contains no start symbol. We sometimes write $\mathcal{P} : E \overset{*}{\rightharpoonup} v$ if $\mathcal{P}$ is a production tree (a derivation tree/a proof tree) for $E \overset{*}{\rightharpoonup} v$. We call a grammar *ambiguous* if, for some E and v, there is more than one production tree for $E \overset{*}{\rightharpoonup} v$. Otherwise, the grammar is *unambiguous*. Note that checking whether an RTG is ambiguous or not is known to be decidable [6].

4.1 Approximation of Evaluation Structure

We construct an RTG from a program so that each production rule in the grammar corresponds to an evaluation step of the program. The basic idea behind the construction has been explained in Sect. 3, and the formal rules are given in Fig. 7. The procedure itself is not new; it is almost the same as the type inference of regular expression types [19], and similar techniques have been adopted in the range inference of tree transducers (e.g., [10]).

To clarify the correspondence between a program and its derived grammar, we define a transformation, approx, from a proof of $\theta \vdash e \downarrow v$ to a proof of $E_e \overset{*}{\rightharpoonup} v$ in Fig. 8. Function approx defines the node-by-node correspondence between the two proofs. Formally, we have the following theorem.

$$\begin{array}{rcl}
x & \longrightarrow & E_x \to \top \\
\mathsf{C}(e_1,\ldots,e_n) & \longrightarrow & E_{\mathsf{C}(e_1,\ldots,e_n)} \to \mathsf{C}(E_{e_1},\ldots,E_{e_n}) \\
f(e_1,\ldots,e_n) & \longrightarrow & E_{f(e_1,\ldots,e_n)} \to E_{e'} \quad \text{where } \exists(f(\ldots)=e') \\
\mathbf{case}\ x\ \mathbf{of}\ \{p_1 \to e_1;\ldots;p_n \to e_n\} & \longrightarrow & \begin{pmatrix} E_{\mathbf{case}\ x\ \mathbf{of}\ \{p_1\to e_1;\ldots;p_n\to e_n\}} \to E_{e_1} \\ \vdots \\ E_{\mathbf{case}\ x\ \mathbf{of}\ \{p_1\to e_1;\ldots;p_n\to e_n\}} \to E_{e_n} \end{pmatrix}
\end{array}$$

Fig. 7. Construction of productions rules of RTG.

$$\begin{aligned}
&\mathsf{approx}\left(\frac{\theta(x)=v}{\theta \vdash x \downarrow v}\right) \\
&= \frac{\overline{\top \xrightarrow{*} v}}{E_x \xrightarrow{*} v} \\
&\mathsf{approx}\left(\frac{\{\mathcal{E}_i : \theta|_{\mathsf{vars}(e_i)} \vdash e_i \downarrow v_i\}_{1\le i\le n}}{\theta \vdash \mathsf{C}(e_1,\ldots,e_n) \downarrow \mathsf{C}(v_1,\ldots,v_n)}\right) \\
&= \frac{\{\mathsf{approx}(\mathcal{E}_i)\}_{1\le i\le n}}{E_{\mathsf{C}(e_1,\ldots,e_n)} \xrightarrow{*} \mathsf{C}(v_1,\ldots,v_n)} \\
&\mathsf{approx}\left(\frac{\{\_ : \theta|_{\mathsf{vars}(e_i)} \vdash e_i \downarrow v_i\}_{1\le i\le n} \quad \mathcal{E} : \{x_i \mapsto v_i \mid 1 \le i \le n\} \vdash e \downarrow v}{\theta \vdash f(e_1,\ldots,e_n) \downarrow v}\right) \\
&= \frac{\mathsf{approx}(\mathcal{E})}{E_{f(e_1,\ldots,e_n)} \xrightarrow{*} v} \qquad \text{where } \exists(f(x_1,\ldots,x_n)=e) \\
&\mathsf{approx}\left(\frac{\exists\sigma,i.\ p_i\sigma = \theta(x) \quad \mathcal{E}_i : (\theta \uplus \sigma)|_{\mathsf{vars}(e_i)} \vdash e_i \downarrow v}{\theta \vdash \mathbf{case}\ x\ \mathbf{of}\ \{p_1 \to e_1;\ldots;p_n \to e_n\} \downarrow v}\right) \\
&= \frac{\mathsf{approx}(\mathcal{E}_i)}{E_{\mathbf{case}\ x\ \mathbf{of}\ \{p_1\to e_1;\ldots;p_n\to e_n\}} \xrightarrow{*} v}
\end{aligned}$$

Fig. 8. Definition of approx.

Theorem 1 (Approximation). *Given evaluation tree $\mathcal{E} : \theta \vdash e \downarrow v$, $\mathcal{P} = \mathsf{approx}(\mathcal{E})$ is a production tree for $E_e \xrightarrow{*} v$, i.e., $\mathcal{P} : E_e \xrightarrow{*} v$.* □

Since approx discards the evaluation trees of arguments at the third branch, approx is neither surjective nor injective: there may be production tree $\mathcal{P} : E_e \xrightarrow{*} v$ that does not correspond to any evaluation tree, i.e., $\forall \mathcal{E} : \theta \vdash e \downarrow v.\ \mathcal{P} \neq \mathsf{approx}(\mathcal{E})$, even if e evaluates to v under some environment. For example, consider the program

$$\begin{aligned}
&h(r) = add(\mathsf{Z}, r) \\
&add(x,r) = \mathbf{case}\ x\ \mathbf{of}\ \{\mathsf{Z} \to r;\ \mathsf{S}(y) \to \mathsf{S}(add(y,r))\}.
\end{aligned}$$

Since h is injective, there is only one evaluation tree for $\{r \mapsto \mathsf{S}(\mathsf{Z})\} \vdash add(\mathsf{Z}, r) \downarrow \mathsf{S}(\mathsf{Z})$. From h we obtain the following grammar:

$$\begin{array}{llllll}
E_{add(\mathsf{Z},r)} & \to E_{\mathbf{case}\ x\ \mathbf{of}\ \{\ldots\}} & E_{add(y,r)} & \to E_{\mathbf{case}\ x\ \mathbf{of}\ \{\ldots\}} & E_y \to \top. \\
E_{\mathbf{case}\ x\ \mathbf{of}\ \{\ldots\}} & \to E_r & E_{\mathsf{Z}} & \to \mathsf{Z} & \\
E_{\mathbf{case}\ x\ \mathbf{of}\ \{\ldots\}} & \to \mathsf{S}(E_{add(y,r)}) & E_r & \to \top &
\end{array}$$

The reader may have found that there are two production trees for $E_{add(\mathsf{Z},r)} \xrightarrow{*} \mathsf{S}(\mathsf{Z})$, and only one of these corresponds to the evaluation tree.

To deal with this situation, we propose two sufficient conditions to guarantee bijection between evaluation and production trees:

Condition (SUFF-LEFT)**:** The program is nonerasing and the derived grammar is unambiguous.

Condition (SUFF-RIGHT)**:** The program is affine and treeless (i.e., every argument of a function call must be a variable) [29].

Roughly speaking, (SUFF-LEFT) guarantees the injectivity of a program, while (SUFF-RIGHT) guarantees its surjectivity with respect to the range described by the grammar. With (SUFF-LEFT), for v of $\mathcal{E} : \theta \vdash e \downarrow v$, any $\mathcal{P} : E_e \xrightarrow{*} v$ must equal $\mathsf{approx}(\mathcal{E})$ since the grammar is unambiguous. With (SUFF-RIGHT), every $\mathcal{P} : E_e \xrightarrow{*} v$ must have a unique corresponding evaluation tree, $\mathcal{E} : \theta \vdash e \downarrow v$ (a direct consequence of [21]). As will be seen later, (SUFF-LEFT) is used to obtain left inverses, and (SUFF-RIGHT) is used to obtain right inverses.

4.2 Reconstructing Evaluation Trees

Our aim now is to construct an evaluation tree from a production tree, i.e., to construct the inverse of approx. Since the RHSs of approx are disjoint, inversion of approx is done in a straightforward way if we can recover the information lost in approx — the evaluation trees of arguments to each function call $f(e_1, \ldots, e_n)$. In other words, in reconstructing the evaluation tree of $f(e_1, \ldots, e_n) \downarrow v$ where f is defined by $f(x_1, \ldots, x_n) = e$, we must recover $\mathcal{E}_i$ of $\mathcal{E}_i : \theta_i \vdash e_i \downarrow \theta(x_i)$ from $\mathcal{E} : \theta \vdash e \downarrow v$. Luckily, this can be done. Assume that each e_i respectively evaluates to v_i. The values of v_i have been recovered by $v_i = \theta(x_i)$. Thus, evaluation tree $\mathcal{E}_i : \theta_i \vdash e_i \downarrow v_i$ is obtained by recursively rebuilding production tree $\mathcal{P}_i : E_{e_i} \xrightarrow{*} v_i$.

Formally, reconst defined in Fig. 9 reconstructs an evaluation tree from a production tree.[5] Function reconst is an inverse of approx obtained by swapping LHSs with RHSs except that invE recovers the lost information of approx, as explained in the previous paragraph. Operator $\uplus$ is extended to substitutions with overlapping domains: $\{x \mapsto 1\} \uplus \{x \mapsto 1\}$ yields $\{x \mapsto 1\}$, while $\{x \mapsto 1\} \uplus \{x \mapsto 2\}$ fails. Note that reconst is a partial function; e.g., $\uplus$ may fail.

Procedure invE in Fig. 9 appears to be nondeterministic since there might be more than one production tree. With constraints (SUFF-LEFT) and (SUFF-RIGHT), we ensure that there is at most one production tree and thus invE is deterministic.

The following properties relate reconst and approx:

[5] For simplicity, we assume that there is at most one possibility to choose p_i at the definition of reconst for **case**; e.g., we exclude **case** x **of** $\{\mathsf{Z} \to \mathsf{Z}; \mathsf{S}(y) \to \mathsf{Z}\}$. Note that a program containing such a **case**-expression does not satisfy (SUFF-LEFT). To invert such programs under (SUFF-RIGHT), it is sufficient to construct a nonterminal for each expression *occurrence* instead of the expression itself in constructing RTG.

$$\mathsf{reconst}\left(\frac{\overline{\top \xrightarrow{*} v}}{E_x \xrightarrow{*} v}\right) = \frac{\{x \mapsto v\}(x) = v}{\{x \mapsto v\} \vdash x \downarrow v}$$

$$\mathsf{reconst}\left(\frac{\{\mathcal{P}_i\}_{1 \le i \le n}}{E_{\mathsf{C}(e_1,\dots,e_n)} \xrightarrow{*} \mathsf{C}(v_1,\dots,v_n)}\right) = \frac{\{\mathcal{E}_i : \theta_i \vdash e_i \downarrow v_i\}_{1 \le i \le n}}{\theta_1 \uplus \cdots \uplus \theta_n \vdash \mathsf{C}(e_1,\dots,e_n) \downarrow \mathsf{C}(v_1,\dots,v_n)} \quad \text{where } (\mathcal{E}_i : \theta_i \vdash e_i \downarrow v_i) = \mathsf{reconst}(\mathcal{P}_i)$$

$$\mathsf{reconst}\left(\frac{\mathcal{P} : E_e \xrightarrow{*} v}{E_{f(e_1,\dots,e_n)} \xrightarrow{*} v}\right) = \frac{\{\mathcal{E}_i : \theta_i \vdash e_i \downarrow v\}_{1 \le i \le n} \quad \mathcal{E} : \theta \vdash e \downarrow v}{\theta_1 \uplus \cdots \uplus \theta_n \vdash f(e_1,\dots,e_n) \xrightarrow{*} v} \quad \text{where } \begin{array}{l} \exists f(x_1,\dots,x_n) = e. \\ (\mathcal{E} : \theta \vdash e \downarrow v) = \mathsf{reconst}(\mathcal{P}) \\ (\mathcal{E}_i : \theta_i \vdash e_i \downarrow v) = \mathsf{invE}(e_i, \theta(x_i)) \end{array}$$

$$\mathsf{reconst}\left(\frac{\mathcal{P}_i : E_{e_i} \xrightarrow{*} v}{E_{\mathbf{case}\ x\ \mathbf{of}\ \{p_1 \to e_1; \dots; p_n \to e_n\}} \xrightarrow{*} v}\right) = \frac{\exists \sigma, i.\ p_i\sigma = \theta(x) \quad \mathcal{E} : \eta \vdash e_i \downarrow v}{\theta \vdash \mathbf{case}\ x\ \mathbf{of}\ \{p_1 \to e_1; \dots; p_n \to e_n\} \downarrow v} \quad \text{where } \begin{array}{l} (\mathcal{E}_i : \eta \vdash e_i \downarrow v) = \mathsf{reconst}(\mathcal{P}_i) \\ \theta = \eta \uplus \{x \mapsto p_i\eta\} \\ \sigma = \eta|_{\mathsf{vars}(p_i)} \end{array}$$

$$\mathsf{invE}(e, v) = \mathsf{reconst}(\mathcal{P}) \quad \text{where } \mathcal{P} \text{ is a production tree of } E_e \xrightarrow{*} v.$$

Fig. 9. Definition of reconst.

Theorem 2. *If* $\mathsf{reconst}(\mathcal{P}) = (\mathcal{E} : \theta \vdash e \downarrow v)$*, then* $\mathcal{E}$ *is a proof of* $\theta \vdash e \downarrow v$.

Proof Sketch. Induction on $\mathcal{P}$. □

Theorem 3. *If* $\mathcal{E} = \mathsf{reconst}(\mathcal{P})$*, then* $\mathsf{approx}(\mathcal{E}) = \mathcal{P}$ *holds.*

Proof Sketch. Induction on $\mathcal{P}$. □

Lemma 1 (Correctness (Left)). *Assume that a program satisfies* (SUFF-LEFT)*. Let* e *be an expression and* v *a value such that* $\exists\theta.\ \theta \vdash e \downarrow v$*. Then, for production tree* $\mathcal{P} : E_e \xrightarrow{*} v$, $\mathsf{reconst}(\mathcal{P}) = (\mathcal{E} : \theta \vdash e \downarrow v)$ *holds.*

Proof Sketch. In this case, $\mathcal{P} = \mathsf{approx}(\mathcal{E} : \theta \vdash e \downarrow v)$ holds for some $\mathcal{E}$ because of the unambiguity of the grammar. Then, by induction on the structure of $\mathcal{E}$, we prove $\mathsf{reconst}(\mathsf{approx}(\mathcal{E})) = \mathcal{E}$, which means $\mathsf{reconst}(\mathcal{P})$ terminates and results in $\mathcal{E}$. The nonerasing property ensures that for each step of reconst, θ of $\theta \vdash e \downarrow v$ is defined for any variable occurring in e. □

Lemma 2 (Correctness (Right)). *Under* (SUFF-RIGHT)*, for any production tree* $\mathcal{P} : E_e \xrightarrow{*} v$, $\mathsf{reconst}(\mathcal{P}) = \mathcal{E} : \theta \vdash e \downarrow v$ *holds.*

Proof Sketch. In this case, since $\mathsf{invE}(e, v)$ is always called with $e = x$, the call terminates and returns an evaluation tree of form $\{x \mapsto v\} \vdash x \downarrow v$. Thus, we conclude that under (SUFF-RIGHT), reconst always terminates. The rest of the proof is straightforward by induction on $\mathcal{P}$. Note that nonerasure does not matter here because we can assign any value to a variable that does not affect the output; leaving it as undefined is a correct solution. □

$$
\begin{array}{l}
\mathit{reverse}'(\mathit{xs}) = \mathit{extract}(\mathit{call}(\mathit{shape}(\mathit{xs}))) \\
\mathit{shape}(\mathit{xs}) = \mathbf{case}\ \mathit{xs}\ \mathbf{of}\ \{[\,] \to \mathsf{Pair}(\mathsf{Z}, [\,]);\ x : \mathit{xs}' \to \mathit{inc}(\mathit{shape}(\mathit{xs}'), x)\} \\
\mathit{inc}(r, x) = \mathbf{case}\ r\ \mathbf{of}\ \{\mathsf{Pair}(n, \mathit{xs}) \to \mathsf{Pair}(\mathsf{S}(n), x : \mathit{xs})\} \\
\mathit{call}(r) = \mathbf{case}\ r\ \mathbf{of}\ \{\mathsf{Pair}(n, \mathit{xs}) \to \mathit{revTABA}(n, \mathit{xs})\} \\
\mathit{extract}(r) = \mathbf{case}\ r\ \mathbf{of}\ \{\mathsf{Pair}(\mathit{xs}, [\,]) \to \mathit{xs}\} \\
\mathit{revTABA}(n, \mathit{xs}) = \mathbf{case}\ n\ \mathbf{of}\ \{\mathsf{Z} \to \mathsf{Pair}(\mathit{xs}, [\,]);\ \mathsf{S}(m) \to \mathit{shift}(\mathit{revTABA}(m, \mathit{xs}))\} \\
\mathit{shift}(r) = \mathbf{case}\ r\ \mathbf{of}\ \{\mathsf{Pair}(x : \mathit{xs}, \mathit{ys}) \to \mathsf{Pair}(\mathit{xs}, x : \mathit{ys})\}
\end{array}
$$

Fig. 10. Variant of *reverse* that is invertible with RTG.

From the lemmas above we can prove the following theorem.

Theorem 4 (Correctness of Grammar-based Inversion). *For a program with definition $f(x_1, \ldots, x_n) = e$, the program f^{-1} defined by*

$$f^{-1}(v) = (\theta(x_1), \ldots, \theta(x_n))\ \mathbf{where}\ (\mathcal{E} : \theta \vdash e \downarrow v) = \mathsf{invE}(e, v)$$

satisfies the following two properties.

1. *f^{-1} is a left inverse of f, if* (Suff-Left) *holds, and*
2. *f^{-1} is a right inverse of f, if* (Suff-Right) *holds.* $\square$

Note that *double* and *snoc* satisfy both (Suff-Left) and (Suff-Right), while *reverse* satisfies neither of them. Program `runlength` used in the experiment to be discussed in Sect. 6 only satisfies (Suff-Left).

Recall that in Sect. 1 we stated that we classify invertible programs instead of problems. We can give another definition of *reverse*, as shown in Fig. 10, from which the derived grammar is unambiguous. More precisely, *reverse*′ satisfies (Suff-Left) but not (Suff-Right). The definition of *reverse*′, while appearing tricky, is nothing but a nonerasing version of IO-swapped *reverse* [22]. Note that both *reverse* and *reverse*′ run in time linear to the input size.

4.3 Properties

We discuss some properties of the inverses derived using grammar-based inversion with RTG.

Correspondence to Post Condition. Post conditions play important roles in many program inversion methods [8,11,16,17]. Post condition P of e, which we write as $e\{P\}$, is a predicate on the state (i.e., values of all free variables) that is supposed to be true after e is executed. A *simple post condition* is a predicate on the value of e. Given a program, one may assign, for each function f, a post condition, post_f. The assignment is *valid* if we can assign a valid post condition to each sub-expression in the program in the way defined below:

Definition 2 (Simple Post Conditions). *Given a program and a post condition assignment for each function in the program, an assignment of post conditions to all sub-expressions is valid if*

- *every variable x is given a post condition, $P(\_) = \mathsf{True}$;*
- *each function call $f(\dots)$ is assigned the post condition, post_f;*
- $\mathsf{C}(e_1\{P_1\}, \dots, e_n\{P_n\})\{P\}$ *is valid if* $\forall i.\ P_i(v_i) \Rightarrow P(\mathsf{C}(v_1, \dots, v_n))$*;*
- $\mathbf{case}\_\,\mathbf{of}\ \{p_1 \to e_1\{P_1\}; \dots; p_n \to e_n\{P_n\}\}\{P\}$ *is valid if* $\exists i.\ P_i(v) \Rightarrow P(v)$*;*
- *in a definition, $f(\dots) = e\{P\}$, the right-hand side is assigned a post condition satisfying $P(v) \Rightarrow \mathsf{post}_f(v)$.*

Many approaches to program inversion rely on disjoint post conditions for each **case** expression. The expression $\mathbf{case}\ x\ \mathbf{of}\ \{p_1 \to e_1\{P_1\}; p_2 \to e_2\{P_2\}\}$, where P_1 and P_2 are disjoint, is inverted to a program that, given output v, tests which of $P_1(v)$ or $P_2(v)$ holds and performs, respectively, the inverse of e_1 or e_2. For non-simple post conditions, it is harder to check the validity of assignment and to test $P(v)$ in executing inverses. Human-assigned post conditions [8,11,17] without validity checks may be more expressive. In contrast, the post conditions in Glück and Kawabe [16] that support inference are basically simple. Note that, in functional language, post conditions can be seen as types satisfying the preservation (subject reduction) law.

The following theorem states that grammar-based inversion using RTG is equivalent to inversion using simple post conditions:

Theorem 5. *The RTG obtained from a program is unambiguous if and only if there exists a valid assignment of simple post conditions such that every* **case***-expression in the program has an assignment*

$$\mathbf{case}\ x\ \mathbf{of}\ \{p_1 \to e_1\{P_1\}; \dots; p_n \to e_n\{P_n\}\}$$

where $P_1 \dots P_n$ are disjoint. That is, for any v, there is at most one P_i such that $P_i(v) = \mathsf{True}$.

Proof Sketch. In this case, for $e\{P\}$, we can prove $(\theta \vdash e \downarrow v) \Rightarrow P(v)$. Then, the "if" part is proved by showing the contraposition: if a grammar is ambiguous, then there exists such a **case**-expression. For grammars obtained with Fig. 7, we can prove that if a grammar is ambiguous, there exists E such that $E \to E_1 \xrightarrow{*} v$ and $E \to E_2 \xrightarrow{*} v$ for distinct E_1 and E_2, and such E must correspond to some **case**-expression. The "only if" part is proved by taking the P of $e\{P\}$ as $P(v) \equiv (\exists \mathcal{P}.\ \mathcal{P} : E_e \xrightarrow{*} v)$. □

It is thus a corollary that to invert more functions than those with grammar-based inversion with RTG, we must use more expressive post conditions that are harder to check, to infer, or to invert.

Efficiency. For RTG, the construction of a production tree for $E \xrightarrow{*} v$ takes time at worst proportional to the size of v [6]. It is remarkable that, thus, if a program is nonerasing, affine, and treeless, the derived inverse runs in $\mathrm{O}(n)$, where n is the size of an input of the inverse. As a result, we can obtain linear time inverses for *double* and *snoc*. Being affine ensures that the domains of environments merged by $\uplus$ are always disjoint; thus, we do not need to spend time checking whether overlapping variables are equal. Being treeless means that arguments e_i of each function call $f(e_1, \ldots, e_n)$ are merely variables. Thus, all production tree constructions at $\mathsf{invE}(x, v)$ immediately match $\top$ with the given value in O(1) time. In more general cases, the construction of production tree $\mathcal{P} : E_e \xrightarrow{*} v$ at $\mathsf{invE}(e, v)$ runs in time between $\mathrm{O}(|\mathcal{P}|)$ and $\mathrm{O}(|v|)$, where $|\mathcal{P}|$ and $|v|$ are the sizes of $\mathcal{P}$ and v, depending on the parsing method. For example, using guided tree automata for parsing [4], we can obtain a linear time inverse for *reverse′* in Fig. 10 because the lower complexity bound is achieved for each call of invE in the inverse. Generally, a derived inverse runs at worst in time that is proportional to the total size of "intermediate data" plus "duplicated data" in addition to the size of the output value. Note that a derived right inverse always takes time at worst linear to the size of its input because (Suff-Right) requires a program to be affine and treeless.

5 Grammar-based Inversion in General

So far, we have discussed grammar-based inversion by RTG as a case study. In this section, we will give more general study on grammar-based inversion.

5.1 More Fine-Grained Classification

Recall that *double* and *snoc* are invertible by RTG. However, the difficulties of inversion differ in the two programs; *double* is easier to invert than *snoc*. Extra conditions for a grammar achieve more fine-grained classification. For example, the grammar of *double* is top-down deterministic (for $E \xrightarrow{*} \mathsf{C}(E_1, \ldots, E_n)$, the tuple $(E_1, \ldots, E_n)$ is unique to E and C) while that of *snoc* is not. If a top-down deterministic grammar has no rule $E \to E_1$ for E that has more than one production rule, swapping LHSs with RHSs results in a deterministic inverse. Even if such production rules exist, additional checking of the root of a value at **case** is sufficient to obtain a deterministic inverse.

5.2 Predefined Inverses as Axioms

Small parts of a program are sometimes very difficult to invert because they use mathematical properties, such as multiplication of prime numbers. In this case, treating them as language constructs with predefined inverses helps us to invert programs that contains them. For example, consider $\mathit{mulPrime}(x_1, x_2)$ that multiplies two primes x_1 and x_2 if $x_1 \leq x_2$. The semantics of function call $\mathit{mulPrime}(e_1, e_2)$ is defined by the predefined semantics $[\![\mathit{mulPrime}]\!]$ as

$$\frac{\{\theta \vdash e_i \downarrow v_i\}_{i=1,2} \quad [\![\mathit{mulPrime}]\!](v_1, v_2) = v \quad v_1 \leq v_2}{\theta \vdash \mathit{mulPrime}(e_1, e_2) \downarrow v} .$$

For the function, we prepare a special production rule, $E_{\mathit{mulPrime}(e_1,e_2)} \to \mathit{Nat}$, where Nat represents natural numbers, and then the corresponding reconst is defined in a straightforward way by using its predefined inverse $[\![\mathit{mulPrime}]\!]^{-1}$.

5.3 More Expressive Grammars

Using more expressive grammars enables us to invert more programs.

Inside-out (IO) context-free tree grammar (CFTG) [9] enables us to investigate accumulation parameters (parameters that are never pattern-matched in evaluation) in parsing. For example, the following IO CFTG can be obtained for *reverse*.

$$\begin{array}{llll} E_{rev(x,[\,])} & \to E_{\textbf{case}\ x\ \textbf{of}\ \{\ldots\}}([\,]) & E_{\textbf{case}\ x\ \textbf{of}\ \{\ldots\}}(r) & \to E_{rev(x,a:r)}(r) \\ E_{\textbf{case}\ x\ \textbf{of}\ \{\ldots\}}(r) & \to r & E_{rev(x,a:r)}(r) & \to E_{\textbf{case}\ x\ \textbf{of}\ \{\ldots\}}(a : r) \end{array}$$

In an RTG, non-terminals do not have parameters/arguments. Thus, as in Sect. 4, when we construct an RTG approximation of a program, we discard the arguments of functions. In an IO CFTG, non-terminals may have "accumulation parameters". Thus, we can similarly construct an IO-CFTG approximation of a program by discarding the arguments that are not accumulation parameters. Note that since approx changes according to the class of grammar, so does its inverse reconst. The change in reconst is straightforward in IO CFTG; similar to reconst for RTG, we re-parse to recover discarded evaluation trees of expressions occurring in non-accumulation parameters. Like (SUFF-LEFT), left-invertible programs are characterized by the unambiguity of the grammar, and like (SUFF-RIGHT), right-invertible programs are characterized by the syntactic condition that ensures that every production tree has a unique corresponding evaluation tree. Note that the class of right-invertible programs by IO CFTG contains the known class of tree transformations called deterministic linear macro tree transducers [10]. IO CFTG corresponds to the post conditions that can contain the variables of accumulation parameters.

For IO CFTG, it is known that checking whether or not $E \xrightarrow{*} v$ holds takes time polynomial to the size of v [3]. Unfortunately, there has been little discussion on "parsing" of IO CFTG because people have not found a use for the production trees. However, we believe that a variant of the CYK parser would yield polynomial-time parsing. Note that, similar to CFG in which a nonterminal generates a string and a string of length n contains n^2 substrings, in IO CFTG, a k-ary nonterminal generates a k-hole context (a value containing k holes to be filled) and a value of size n contains n^{k+1} k-hole contexts. Thus, we believe memoization as in CYK parsing should be applicable. For the example of *reverse*, we can obtain a linear-time inverse by using deterministic bottom-up push-down tree automata [28]. We also believe that it is possible to use a more expressive grammar, e.g., supporting equality check or synchronous production as in the tupled [20] function. For the string case, these features are adopted without violating the polynomial-time-parisible property [5]. Note that, even if the derived RTG is unambiguous, when a derived IO CFTG is can be parsed in linear time, the inverse derived by IO CFTG is sometimes more efficient than that derived by RTG. An inverse derived by IO CFTG calls invE no more than that derived by RTG; the inverse does not call invE for arguments occurring at accumulation-parameter positions of a function call because the evaluation has already been captured by the IO CFTG.

Note that ambiguity check for a grammar beyond regular, such as IO CFTG, is usually undecidable [18]. However, some automated systems or some restricted forms of programs can still guarantee the unambiguity of an expressive grammar in some cases. Investigations into appropriate ways to define "some" programs for which the ambiguity check of derived grammars are decidable would be important in future work.

6 Experiment

This section reports our automatic inversion system[6] using Haskell, and explains that the overhead of the derived inversion to the handwritten inverse is acceptable through an experiment with the implementation. The acceptably-small overhead revealed that our method is not only theoretically feasible but also useful for implementing a program inversion system for acceptably-efficient inverses. Note that to derive inverses as efficient as possible is not our main issue, but this is important because it is a general issue with program inversion.

6.1 Implementation

The prototype system implements grammar-based inversion with RTG (Sect. 4). The system takes a program, and then generates a Haskell program of the left inverse if (Suff-Left) holds. Otherwise, the system generates a Haskell program, which becomes a right inverse of the program if (Suff-Right) holds.

For parsing, the implementation uses guided tree automata [4], allowing $\top$ to match any value. Since a guided tree automaton performs a top-down traversal before a bottom-up traversal, the special case for $\top$ is easy to implement. The derived inverse does not construct production or evaluation trees; they are eliminated by program fusion. Recall that what we need is only θ of $\theta \vdash e \downarrow v$ for given e and v. The implementation determinizes tree automata to reduce the overhead caused by nondeterminism of parsing. Although determinization costs $O(2^n)$, where n is the size of an automaton ($\simeq$ the size of a program), this cost is not severe for our purposes at least for the programs we tested in the experiments.

6.2 Comparison with Handwritten Inverses

For several programs, we compared the execution time of automatically-derived (left) inverses and handwritten (left) inverses for large inputs.[7] Three programs were investigated in the experiment: `snoc` and `double` implement *snoc* and *double* in Sect. 1, respectively, and `runlength` implements run-length encoding as in Fig. 11. Note that, for these three programs, the system can derive an

[6] Available on: `http://www.ipl.t.u-tokyo.ac.jp/~kztk/PaI/`.
[7] We used a PC with an Intel Core2 E8400 (3 GHz) CPU and 2-GB memory, and used Haskell compiler GHC 6.8.2.

```
$runlength(x) = \textbf{case}\ x\ \textbf{of}\ \{[\,] \rightarrow [\,];\ a : y \rightarrow step(runlength(y), a)\}$
$step(x, a) = \textbf{case}\ x\ \textbf{of}\ \{\ [\,]$                 $\rightarrow \mathsf{Pair}(a, zero()) : [\,];$
                              $\mathsf{Pair}(b, n) : y \rightarrow updateRL(eq(a, b), n, y)\ \}$
$updateRL(i, n, y) = \textbf{case}\ i\ \textbf{of}\ \{\ \mathsf{Right}(a) \rightarrow \mathsf{Pair}(a, inc(n)) : y;$
                                 $\mathsf{Left}(a, b) \rightarrow \mathsf{Pair}(a, zero()) : \mathsf{Pair}(b, n) : y\ \}$
...
```

Fig. 11. `runlength`: $eq(a, b)$ returns $\mathsf{Right}(a)$ if $a = b$, otherwise returns $\mathsf{Left}(a, b)$, and $zero()$ and $inc(x)$ are 0 and +1 on binary representation of numbers, respectively. Here $\mathsf{Pair}(a, n)$ means $inc(n)$-times successive occurrences of a.

Table 1. Results of experiment.

Program	Inversion (s)	#Input	Automatically-Derived (s)	Handwritten (s)
`snoc`	< 0.05	$\simeq$ 8 millions	0.95	0.67
`double`	< 0.05	$\simeq$ 10 millions	0.23	0.11
`runlength`	0.3	$\simeq$ 9 millions	0.76	0.33

inverse that has the same complexity as that of a handwritten inverse because construction of production tree $\mathcal{P} : E_e \xrightarrow{*} v$ at $\mathsf{invE}(e, v)$ runs in $\mathsf{O}(|\mathcal{P}|)$.

All of these three programs satisfy (Suff-Left). The results of the experiment are listed in Table 1. Each column represents the following: Program denotes the investigated program, Inversion denotes the elapsed time for inversion including code generation, #Input denotes the number of constructors occurring in the input tree, and Automatically-Derived and Handwritten denote the elapsed time of the automatically-derived inverse and the handwritten inverse, respectively. The size of input for each pair of automatically-derived and handwritten inverses was chosen to enable the elapsed time to compared in seconds, as long as there was no shortage of memory.

Inversion in Table 1 indicates that our implemented inversion runs very efficiently. Even though in `runlength` the inversion process took about 0.3 seconds, we found by extra profiling that more than half the time was spent for serialization that makes a textual code from an abstract syntax tree. Automatically-Derived and Handwritten in Table 1 indicate that the derived inverses run from a half to a third of the speed of the handwritten inverses. We believe that this small ratio would be acceptable. In addition, we expect that the ratio can scale because the ratios of small programs such as `snoc` and `double` are almost the same as that of a relatively involved program such as `runlength`.

7 Related Work

Many approaches to program inversion have been proposed [1, 7, 8, 11, 15–17, 25, 27, 30]. These methods are based on reverting the execution order of an input program, unlike our method. Of these, those by Yellin [30] and Glück and Kawabe [15] are the most closely related to ours.

Yellin [30] inverted string-to-string transformations written in a restricted class of attribute grammars. His idea is an extension of evaluation of synchronous grammars [2] — transformation by using two CFGs that share the same parse tree modulo permutation of children. We borrowed his basic idea of restoring the evaluation structure by parsing. Instead of CFG, we used tree grammars because functional programs describe tree transformations. Regarding of the class of invertible functions, with the restricted class of AG, one cannot deconstruct intermediate results, ruling out programs like `runlength`, i.e., those programs are not handled by his approach. His framework, on the other hand, is more suitable for programs defined using **if**-expressions, while we handle them indirectly as in the *eq* and *updateRL* in Fig. 11.

Glück and Kawabe [15] constructed inverse programs by reversing programs, before applying LR-parsing to the derived sequential programs to resolve nondeterminism. While our method and theirs are both "grammar-based", they place more emphasis on obtaining efficient inverses. Their method consists of the three steps: (1) convert a program to a program in their stack-based language, (2) apply LR-parsing to the stack-based program by taking the program to be CFG, and (3) generate a program in which the stack in LR-parsing is emulated by the stack of function call. Due to these three steps, they obtained the efficient inverses because the inverses that have no parsing overhead. However, what class of programs is invertible is less clear in theirs because all three steps affect invertibility. Steps 2 and 3 may fail, and Step 1 affects the later steps because, for non-linear recursive functions, the result of Step 1 differs if we choose a different evaluation order. Examples of functions discussed in this paper can be handled with their method, while many of the programs they handled would be invertible by grammar-based inversion with IO CFTG using a deterministic bottom-up push-down tree automaton [28], which is a counterpart of "LR-parsing" in CFTG. Theoretically, even with RTG, there exist programs that can be handled by ours but not theirs.

Many program inversion techniques rely on proof of injectivity. In many existing approaches, post conditions [8,11,16,17] for branching statements/expressions are used for this purpose. In Nishida and Sakai [25], completion is used to check whether the obtained nondeterministic program is actually a function, which implies the injectivity of the original program [26]. In grammar-based inversion, we check injectivity by checking the unambiguity of grammar.

Another way to obtain inverse programs is, similarly to combinator-based bidirectional language [12], to construct programs using invertible combinators [23]; a program constructed in this way comes together with its inverse. Our method can be incorporated into such combinator-based frameworks both for providing basic invertible combinators and for gluing combinators as in Sect. 5.2. In their frameworks [12, 23], accumulative functions such as *reverse* cannot be represented directly but must be written as $reverse'$ in Sect. 4. We believe that grammar-based inversion with grammar beyond RTG would enable us to invert more functions that are written in more natural forms.

Abramov and Glück [1] categorized inversion methods into *program inversion* and *inverse computation*. Program inversion takes a program and returns an inverse program while inverse computation takes a program and an output and computes the corresponding input. The two methods are different in two points: A minor difference is that program inversion performs code generation, but the main difference is the existence of partial evaluation; i.e., in program inversion the obtained inverse is specialized to the input program. Note that the two notions are not so different theoretically because generating a program that simply calls "eval" to the pair of an inverse computation and an input program achieves program inversion. Thus, it is important to discuss how much the inverse computation is specialized to the input program. In grammar-based inversion, the main chance of partial evaluation is when parsing the grammar. For a grammar derived statically for an input program, we can choose an appropriate parsing method according to the characteristics of the grammar.

The tree transducer [14] is a family of formal models of tree transformation. Instances of tree transducers vary in terms of expressive power and difficulty of inversion. We did not use tree transducers because, in all models of tree transducers we are aware of, a function may not perform case analysis on the output of another function, while many programs we are interested in (e.g. `runlength` in Fig. 11) are of the form $g(\ldots) = f(g(x))$ with an invertible f that looks into the result of g. We did, however, borrow many ideas from tree transducers, e.g., the grammar construction in Sect. 4.

8 Conclusion

We proposed grammar-based inversion, which is a framework for program inversion. Grammar-based inversion can describe how difficult inverting a program is through the complexity of the unambiguous grammar used for inversion. At the same time, the complexity of parsing determines the worst-case complexity of a derived inverse.

Grammar-based inversion gives us a new view of program inversion. With it, we can split program inversion into two problems: finding an unambiguous grammar that captures the evaluation structure of a program, and finding an efficient parsing method for the grammar. For example, so far, many inversion methods except Glück and Kawabe [15] have not handled functions containing accumulation parameters. A solution with grammar-based inversion for such functions is to use grammar such as IO CFTG that can capture the accumulation structure, and to find an efficient parsing method specialized to the grammar.

Although grammar-based inversion can derive a right inverse, this is not very useful because, in many applications, users do not want an arbitrary right inverse but *some* right inverse. That is, some right inverse is more preferable than other right inverses. For example, a right inverse achieving a high compression rate is preferable in LZSS compression where a compression procedure is a right inverse of decompression. Another interesting example is bidirectional transformation (e.g., [12]). In bidirectional transformation, function $f :: S \to V$ is coupled with its backward semantics $f_B :: (S, V) \to S$; if the result of f is changed from $f(s)$ to v, the change is put back on S as the change from s to $f_B(s, v)$. A simple example of bidirectional transformation is component extraction from a tuple, such as $\mathit{fst}(s_1, s_2) = s_1$ coupled with $\mathit{fst}_B((s_1, s_2), v) = (v, s_2)$. In bidirectional transformation, backward semantics f_B is a right inverse of its forward semantics f if the first argument of f_B is fixed. In such right inverses, a right inverse that achieves as small modification as possible is often preferable. It would be important to extend the framework to accept user-defined "preferable" measures to make grammar-based inversion more applicable.

Acknowledgments. We would like to thank Robert Glück, Keisuke Nakano, Kazuhiro Inaba, and Akimasa Morihata for their valuable comments on an earlier version of the paper. We are also grateful to the anonymous reviewers who gave us helpful advice to improve the paper. This work is partially supported by the Japan Society for the Promotion of Science, Grant-in-Aid for JSPS Fellows $20 \cdot 9584$ and Grant-in-Aid for Scientific Research (A) 19200002, and the Grand-Challenging Project on "Linguistic Foundation for Bidirectional Model Transformation" from National Institute of Informatics.

References

1. Abramov, S.M., Glück, R.: Principles of Inverse Computation and the Universal Resolving Algorithm. In: The Essence of Computation, pp. 269–295. (2002)
2. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1972)
3. Asveld, P.R.J.: Time and Space Complexity of Inside-Out Macro Languages. Int. J. Comput. Math. 10(1), 3–14 (1981)
4. Biehl, M., Klarlund, N., Rauhe, T.: Algorithms for Guided Tree Automata. In: Raymond, D.R., Wood, D., Yu, S. (eds.) WIA '96. LNCS, vol. 1260, pp. 6–25. Springer, Heidelberg (1997)
5. Boullier, P.: Range Concatenation Grammars. In: New Developments in Parsing Technology. Kluwer Academic Publishers, Norwell, MA, USA. pp. 269–289 (2004)
6. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`
7. Dershowitz, N., Mitra, S.: Jeopardy. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 16–29. Springer, Heidelberg (1999)
8. Dijkstra, E.W.: Program Inversion. In: Program Construction. LNCS, vol. 69, pp. 54–57. Springer, Heidelberg (1978)
9. Engelfriet, J., Schmidt, E.M.: IO and OI. I. J. Comput. Syst. Sci. 15(3), 328–353 (1977)
10. Engelfriet, J., Vogler, H.: Macro Tree Transducers. J. Comput. Syst. Sci. 31(1), 71–146 (1985)
11. Eppstein, D.: A Heuristic Approach to Program Inversion. In: International Joint Conference on Artificial Intelligence (IJCAI-85), pp. 219–221. (1985)
12. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. ACM Trans. Program. Lang. Syst. 29(3), (2007)

13. Frisch, A.: Regular Tree Language Recognition with Static Information. In: Lévy, J.J., Mayr, E.W., Mitchell, J.C. (eds.) Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), pp. 661–674. Kluwer (2004)
14. Fülöp, Z., Vogler, H.: Syntax-Directed Semantics: Formal Models Based on Tree Transducers. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1998)
15. Glück, R., Kawabe, M.: A Method for Automatic Program Inversion Based on LR(0) Parsing. Fundam. Inform. 66(4), 367–395 (2005)
16. Glück, R., Kawabe, M.: Revisiting an Automatic Program Inverter for Lisp. SIGPLAN Notices. 40(5), 8–17 (2005)
17. Gries, D.: 21 Inverting Programs. In: The Science of Programming. Springer (1981)
18. Hopcroft, J.E., Motwani, R., Ullman, J.D.: 7 Properties of Context-Free Languages. In: Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc. (2006)
19. Hosoya, H., Pierce, B.C.: XDuce: A Statically Typed XML Processing Language. ACM Trans. Internet Techn. 3(2), 117–148 (2003)
20. Hu, Z., Iwasaki, H., Takeichi, M., Takano, A.: Tupling Calculation Eliminates Multiple Data Traversals. In: ICFP '97: Proceedings of the second ACM SIGPLAN International Conference on Functional Programming, pp. 164–175. ACM Press, New York, NY, USA (1997)
21. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization Transformation based on Automatic Derivation of View Complement Functions. In: ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, pp. 47–58. ACM, New York, NY, USA (2007)
22. Morihata, A., Kakehi, K., Hu, Z., Takeichi, M.: Swapping Arguments and Results of Recursive Functions. In: Uustalu, T. (ed.) MPC 2006. LNCS, vol. 4014, pp. 379–396. Springer, Heidelberg (2006)
23. Mu, S.C., Hu, Z., Takeichi, M.: An Injective Language for Reversible Computation. In: Kozen, D., Shankland, C. (eds.) MPC 2004. LNCS, vol. 3125, pp. 289–313. Springer, Heidelberg (2004)
24. Neumann, A., Seidl, H.: Locating Matches of Tree Patterns in Forests. In: Arvind, V., Ramanujam, R. (eds.) FSTTCS 1998. LNCS, vol. 1530, pp. 134–145. Springer, Heidelberg (1998)
25. Nishida, N., Sakai, M.: Completion after Program Inversion of Injective Functions. Electr. Notes Theor. Comput. Sci. 237, 39–56 (2009)
26. Nishida, N., Sakai, M.: Proving Injectivity of Functions via Program Inversion in Term Rewriting. accepted in FLOPS 2010. (2010)
27. Nishida, N., Sakai, M., Sakabe, T.: Partial Inversion of Constructor Term Rewriting Systems. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 264–278. Springer, Heidelberg (2005)
28. Schimpf, K.M., Gallier, J.H.: Tree Pushdown Automata. J. Comput. Syst. Sci. 30(1), 25–40 (1985)
29. Wadler, P.: Deforestation: Transforming Programs to Eliminate Trees. Theor. Comput. Sci. 73(2), 231–248 (1990)
30. Yellin, D.M.: Attribute Grammar Inversion and Source-to-Source Translation. LNCS, vol. 302. Springer, Heidelberg (1988)