Higher-Order Symb Comput
DOI 10.1007/s10990-013-9097-8

Polynomial-time inverse computation for accumulative functions with multiple data traversals

Kazutaka Matsuda · Kazuhiro Inaba · Keisuke Nakano

Abstract The problem of inverse computation has many potential applications such as serialization/deserialization, providing support for undo, and test-case generation for software testing. In this paper, we propose an inverse computation method that always terminates for a class of functions known as parameter-linear macro tree transducers, which involve multiple data traversals and the use of accumulations. The key to our method is the observation that a function in the class can be regarded as a non-accumulative context-generating transformation without multiple data traversals. Accordingly, we demonstrate that it is easy to achieve terminating inverse computation for the class by context-wise memoization of the inverse computation results. We also show that when we use a tree automaton to express the inverse computation results, the inverse computation runs in time polynomial to the size of the original output and the textual program size.

Keywords Program inversion · Inverse computation · Program transformation · Functional programming · Tree automata · Tree transducers

A preliminary summary of this article appeared in Proceedings of PEPM 2012, pp. 5–14, 2012.

K. Matsuda (✉)
Graduate School of Information Science and Technology, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo, 113-0033 Japan
e-mail: kztk@is.s.u-tokyo.ac.jp

K. Inaba
National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430 Japan
e-mail: kiki@kmonos.net

Present address:
K. Inaba
Google Inc., PO Box 22, 6-10-1 Roppongi, Minato-ku, Tokyo, 106-6126 Japan

K. Nakano
Center for Frontier Science and Engineering, The University of Electro-Communications, 1-5-1 Chofugaoka, Chofu, Tokyo, 182-8585 Japan
e-mail: ksk@cs.uec.ac.jp

Published online: 25 September 2013

1 Introduction

The problem of inverse computation [1, 2, 20, 21, 23, 30, 37, 40, 49]—finding an input s for a program f and a given output t such that $f(s) = t$—has many potential applications, including test-case generation in software testing, supporting undo/redo, and obtaining a deserialization from a serialization program.

Let us illustrate the problem with an example. Suppose that we want to write an evaluator for a simple arithmetic expression language defined by the following datatype. (We basically follow the Haskell syntax [6] even though we target an untyped first-order functional language with call-by-value semantics.)

$$\begin{aligned}&\textbf{data}\ \mathit{Val} \ = \mathsf{Z} \mid \mathsf{S}(\mathit{Val})\\ &\textbf{data}\ \mathit{Exp} = \mathsf{Zero} \mid \mathsf{One} \mid \mathsf{Add}(\mathit{Exp}, \mathit{Exp}) \mid \mathsf{Dbl}(\mathit{Exp})\end{aligned}$$

Informally, Z and Zero represent 0, One represents 1, $\mathsf{S}(n)$ means $n+1$ (the successor of n), $\mathsf{Add}(n_1, n_2)$ adds the numbers n_1 and n_2, and $\mathsf{Dbl}(n)$ doubles the number n.

An evaluator $\mathit{eval} :: \mathit{Exp} \to \mathit{Val}$ of the expressions can be implemented as follows.

$$\begin{aligned}&\mathit{eval}(x) = \mathit{evalAcc}(x, \mathsf{Z})\\ &\\ &\mathit{evalAcc}(\mathsf{Zero}, y) \ \ = y\\ &\mathit{evalAcc}(\mathsf{One}, y) \ \ = \mathsf{S}(y)\\ &\mathit{evalAcc}(\mathsf{Add}(x_1, x_2), y) = \mathit{evalAcc}(x_1, \mathit{evalAcc}(x_2, y))\\ &\mathit{evalAcc}(\mathsf{Dbl}(x), y) \ \ = \mathit{evalAcc}(x, \mathit{evalAcc}(x, y))\end{aligned}$$

Here, *eval* uses *evalAcc* that uses *accumulations*. The function *evalAcc* satisfies the invariant that $\mathit{evalAcc}(e, m) = \mathit{eval}(e) + m$, where "+" is the addition operator for values. This invariant enables us to read the definition intuitively; *e.g.*, the case of Dbl can be read as $\mathit{eval}(\mathsf{Dbl}(x)) + y = \mathit{eval}(x) + \mathit{eval}(x) + y$.

The inverse computation of *eval*, which enumerates the inputs $\{s \mid \mathit{eval}(s) = t\}$ for a given t, is sometimes useful for testing computations on E. For example, suppose that we write an optimizer f that converts all the expressions e satisfying $\mathit{eval}(e) = \mathsf{S}^{2^n}(\mathsf{Z})$ into $\mathsf{Dbl}^n(\mathsf{One})$, and we want to test if the optimizer works correctly or not, *i.e.*, whether $\mathit{eval}(e) = \mathsf{S}^{2^n}(\mathsf{Z})$ implies $f(e) = \mathsf{Dbl}^n(\mathsf{One})$ or not.[1] A solution would involve randomly generating or enumerating expressions e, filtering out the es that do not satisfy $\mathit{eval}(e) = \mathsf{S}^{2^n}(\mathsf{Z})$, and checking $f(e) = \mathsf{Dbl}^n(\mathsf{One})$. However, it is unsatisfactory because it is inefficient; the majority of the expressions do not evaluate to $\mathsf{S}^{2^n}(\mathsf{Z})$. Inverse computation enables us to generate only the test-cases that are relevant to the test. A test with inverse computation can be efficiently performed by (1) picking up a number m of the form $\mathsf{S}^{2^n}(\mathsf{Z})$, (2) picking up an expression e from the set obtained from the inverse computation for m, and (3) checking if the optimizer f converts e into $\mathsf{Dbl}^n(\mathsf{One})$. Here, all the picked up (randomly generated or enumerated) data are relevant to the final check in the Step (3). Lazy SmallCheck [43] and EasyCheck [9] use inverse computation for efficient test-case generation, which of course has to be supported by efficient inverse computation.

However, there are as yet no systematic efficient inverse computation methods that can handle *eval*. One reason is that *evalAcc* contains *accumulations* and *multiple data traversals*. It is so far unclear how to perform tractable terminating inverse computation for functions

[1] We use the shorthand notation $g^n(x)$ to stand for $\underbrace{g(\ldots(g(x))\ldots)}_{n}$.

with accumulations and multiple data traversals (Sect. 2). Some of the existing methods [1, 2, 21, 37] do not terminate for functions with accumulations. Some approaches [20, 39, 40] can handle certain accumulative computations efficiently, but they do not work for non-injective functions such as *eval*. Although some inverse computation methods terminate for accumulative functions [17, 32], the complexity upper bound is unclear when there are also multiple data traversals.

In this paper, we propose an inverse computation method that can handle a class of accumulative functions like *eval* that have multiple data traversals, namely deterministic macro tree transducers [15] with the restriction of *parameter-linearity* (Sect. 3). In this class of functions, variables for accumulation (such as y in *evalAcc*) cannot be copied but inputs (such as x, x_1 and x_2 in *evalAcc*) can be traversed in many times (as x). Our method computes the set $\{s \mid f(s) = t\}$ as a tree automaton [10] for a given function f and an output y in time polynomial to the size of y (Sect. 4). The key to our inverse computation is the observation that a program in the parameter-linear macro tree transducers is indeed a non-accumulative transformation that generates contexts (*i.e.*, trees with holes) without multiple data traversals. From this viewpoint, we can do the inverse computation through a variant of the existing inverse computation methods [1, 2, 4]. Note that viewing a program as a context-generating transformation is not new. What is new in our paper is to use this view to achieve polynomial-time inverse computation for the class of accumulative functions with multiple data traversals.

Our main contributions are summarized as follows.

- We demonstrate that simply viewing a function as a context-generating transformation helps us to achieve a systematic inverse computation method for accumulative functions. After converting a program into a context-generating one, it is easy to perform inverse computation for the program.
- We show that, for parameter-linear macro tree transducers, our inverse computation method runs in time polynomial to the size of the output and the textual program size, and in time exponential to the number of the functions in the program.

The rest of the paper is organized as follows. Section 2 shows an overview of our proposal. Section 3 defines the target language, parameter-linear macro tree transducers. Section 4 formally presents our inverse computation method. Section 5 reports and discusses the experimental results with our prototype implementation. Section 6 shows four extensions of our proposal, and Sect. 7 shows the relationship between ours and the other research. Section 8 concludes the paper and outlines future work.

The preliminary version of this article has appeared in [36]. The main difference from the version is that we have implemented the proposed algorithm and performed some experiments (Sect. 5). We also have added discussions on the two further extensions of our proposed method (Sects. 6.3 and 6.4) and some explanations in several places.

2 Overview

In this section, we give a brief overview of our proposal.

2.1 Review: when inverse computation terminates

Let us begin with an illustrative example showing when a simple inverse computation [1, 4] terminates. The following function *parity* takes a natural number n and returns n mod 2.

$$\begin{array}{ll} \mathit{parity}(\mathsf{Z}) & = \mathsf{Z} \\ \mathit{parity}(\mathsf{S}(x)) & = \mathit{aux}(x) \\ \\ \mathit{aux}(\mathsf{Z}) & = \mathsf{S}(\mathsf{Z}) \\ \mathit{aux}(\mathsf{S}(x)) & = \mathit{parity}(x) \end{array}$$

What should we do for inverse computation of *parity* given an original output t? Abramov and Glück [1, 2] used a symbolic computation method called (needed) narrowing[2] [4] as a simple way to find a substitution θ such that $\mathit{parity}(x)\theta \stackrel{?}{=} t$, where $\stackrel{?}{=}$ represents an equivalence check of (first-order) values defined in a standard way (*e.g.*, $\mathsf{Z} \stackrel{?}{=} \mathsf{Z} \equiv \top$). The same idea is also shared among logic programming languages such as Curry and Prolog.[3] Roughly speaking, a narrowing is a substitution followed by a reduction, and it can reduce an expression with free variables. For example, $\mathit{parity}(x)$ is not reducible, but, if we substitute Z to x, we can reduce the expression to Z. Such a reduction after a substitution is a narrowing that can be written as $\mathit{parity}(x) \leadsto_{x \mapsto \mathsf{Z}} \mathsf{Z}$. The notion can naturally be extended to equivalence checks, such as $(\mathit{parity}(x) \stackrel{?}{=} \mathsf{Z}) \leadsto_{x \mapsto \mathsf{Z}} (\mathsf{Z} \stackrel{?}{=} \mathsf{Z}) \equiv \top$. By using narrowing, we can obtain the corresponding input by collecting the substitutions used in the narrowing. For example, consider the inverse computation of *parity* for an output Z. Since we have[4]

$$(\mathit{parity}(x) \stackrel{?}{=} \mathsf{Z}) \leadsto_{x \mapsto \mathsf{Z}} \top$$

we know that $\mathit{parity}(\mathsf{Z}) = \mathsf{Z}$, and since we have

$$(\mathit{parity}(x) \stackrel{?}{=} \mathsf{Z}) \leadsto_{x \mapsto \mathsf{S}(x)} (\mathit{aux}(x) \stackrel{?}{=} \mathsf{Z}) \leadsto_{x \mapsto \mathsf{S}(x)} (\mathit{parity}(x) \stackrel{?}{=} \mathsf{Z}) \leadsto_{x \mapsto \mathsf{Z}} \top$$

we know that $\mathit{parity}(\mathsf{S}(\mathsf{S}(\mathsf{Z}))) = \mathsf{Z}$.

Sometimes, the simple inverse computation does not terminate; this happens especially when we give it an output that has no corresponding inputs. For example, the simple inverse computation of *parity* for an output $\mathsf{S}(\mathsf{S}(\mathsf{Z}))$ runs infinitely:

$$\begin{aligned} (\mathit{parity}(x) \stackrel{?}{=} \mathsf{S}^2(\mathsf{Z})) &\leadsto_{x \mapsto \mathsf{S}(x)} (\mathit{aux}(x) \stackrel{?}{=} \mathsf{S}^2(\mathsf{Z})) \\ &\leadsto_{x \mapsto \mathsf{S}(x)} (\mathit{parity}(x) \stackrel{?}{=} \mathsf{S}^2(\mathsf{Z})) \leadsto_{x \mapsto \mathsf{S}(x)} \cdots \end{aligned}$$

One might notice that the check $(\mathit{parity}(x) \stackrel{?}{=} \mathsf{S}^2(\mathsf{Z}))$ occurs twice in the sequence.

Actually, with memoization, the simple inverse computation for *parity* always terminates. For the above narrowing sequence, by memoizing all the checks in the sequence, we can tell that the same check $(\mathit{parity}(x) \stackrel{?}{=} \mathsf{S}^2(\mathsf{Z}))$ occurs twice, and hence the narrowing sequence cannot produce any result. In general, the number of equality checks occurring in the inverse computation is finite because it always has the form $f(x) \stackrel{?}{=} t$ (up to α-renaming), where t is a subterm of the original output given to the inverse computation. Thus, the simple inverse computation always terminates with memoization for *parity*.

This observation also gives an upper bound of the worst-case complexity of inverse computation of *parity*; it runs in constant time regardless the size of the original output because

[2] Precisely speaking, they used *driving* [45] instead of narrowing; these notions are developed independently in the different context, but they essentially have the same mechanism [3].

[3] See [22] for the correspondence between driving and SLD-resolution.

[4] Here, we implicitly apply the reduction rules of $\stackrel{?}{=}$ as much as possible.

the checks in the narrowing have the form of either $parity(x) \stackrel{?}{=} t$ or $aux(x) \stackrel{?}{=} t$, only where t is the original output.

2.2 Problem: non-termination due to accumulations and multiple data traversals

Consider a simplified version of *eval*:

$$\begin{aligned} ev(x) &= evA(x, \mathsf{Z}) \\ evA(\mathsf{One}, y) &= \mathsf{S}(y) \\ evA(\mathsf{Add}(x_1, x_2), y) &= evA(x_1, evA(x_2, y)) \\ evA(\mathsf{Dbl}(x), y) &= evA(x, evA(x, y)) \end{aligned}$$

Though simplified, this function still contains the challenging issues: accumulations and multiple data traversals. Since we have $\{s \mid ev(s) = \mathsf{S}^2(\mathsf{Z})\} = \{\mathsf{Dbl}(\mathsf{One}), \mathsf{Add}(\mathsf{One}, \mathsf{One})\}$ for example, the inverse computation of *ev* for $\mathsf{S}^2(\mathsf{Z})$ should result in the set above.

Unlike *parity*, the simple inverse computation method does not always terminate. For example, the simple inverse computation of *ev* for Z does not terminate.

$$\begin{aligned} (ev(x) \stackrel{?}{=} \mathsf{Z}) &\rightsquigarrow (evA(x, \mathsf{Z}) \stackrel{?}{=} \mathsf{Z}) \\ &\rightsquigarrow_{x \mapsto \mathsf{Dbl}(x)} (evA(x, evA(x, \mathsf{Z})) \stackrel{?}{=} \mathsf{Z}) \rightsquigarrow_{x \mapsto \mathsf{Dbl}(x)} \cdots \end{aligned}$$

Memoization is no longer useful for making the simple inverse computation terminate because there are no repeated checks in the infinite sequence.

The following issues make it difficult for the inverse computation to terminate and even harder to run it in polynomial time.

- *Accumulations*, a sort of call-time computation commonly used in tail recursion, increase the size of the terms in the narrowing process. For example, *evA* contains the accumulations

  $$evA(\mathsf{Dbl}(x), y) = evA(x, \underline{evA(x, y)})$$

  which increase the term-size in the following narrowing steps.

  $$(evA(x, \underline{\mathsf{Z}}) \stackrel{?}{=} \mathsf{Z}) \rightsquigarrow_{x \mapsto \mathsf{Dbl}(x)} (evA(x, \underline{evA(x, \mathsf{Z})}) \stackrel{?}{=} \mathsf{Z})$$

  We can see that the second argument of *evA* (underlined above) gets bigger in narrowing.
- *Multiple data traversals* make things much worse. It prevents us from considering function calls separately. For example, we have to track the two calls $\underline{evA(x, \underline{evA(x, y)})}$ simultaneously. We can see that the number of function calls we have to track simultaneously increases in narrowing. To clarify the problem caused by multiple data traversals, we will look at the issue of accumulations in the absence of multiple data traversals. Suppose that *ev* does not have the case for Dbl and thus does not contain multiple data traversals. Although there are still an infinite narrowing sequence

  $$\begin{aligned} (ev(x) \stackrel{?}{=} \mathsf{Z}) &\rightsquigarrow (evA(x, \mathsf{Z}) \stackrel{?}{=} \mathsf{Z}) \\ &\rightsquigarrow_{x \mapsto \mathsf{Add}(x_1, x_2)} (evA(x_1, evA(x_2, \mathsf{Z})) \stackrel{?}{=} \mathsf{Z}) \rightsquigarrow \ldots, \end{aligned}$$

one can make the simple inverse computation terminate by decomposing the check $(evA(x_1, evA(x_2, \mathsf{Z})) \stackrel{?}{=} \mathsf{Z})$ into $evA(x_1, z) \stackrel{?}{=} \mathsf{Z} \wedge evA(x_2, \mathsf{Z}) \stackrel{?}{=} z$ and by observing that, for $evA(x_1, z) \stackrel{?}{=} z'$, we only need to consider the substitutions that map z and z' to subterms of the output fed to the inverse computation, *i.e.*, Z. Thus, we can substitute a concrete subterm to z and check $evA(x_1, \mathsf{Z}) \stackrel{?}{=} t$ and $evA(x_2, t) \stackrel{?}{=} \mathsf{Z}$ *separately* for a concrete t (a more refined idea can be found in [17, 32]), and we can bound the complexity of inverse computation in a similar way as we did for *parity*. However, this idea does not scale for functions with multiple data traversals, in which many function calls are tracked simultaneously in narrowing. Although the existing approaches [17, 32] achieve terminating inverse computation of certain accumulative functions with multiple data traversals, it is unclear whether there are polynomial-time inverse computations for functions with multiple data traversals.

2.3 Our idea

One might have noticed that the result of $evA(s, t)$ can be written as $K_s[t]$ whatever t is, where K_s is a context (*i.e.*, a tree with holes like $\mathsf{S}(\bullet)$) determined by s and $K_s[t]$ is the tree obtained from K_s by replacing $\bullet$ with t. For example, we have $evA(\mathsf{One}, \underline{\mathsf{Z}}) = \mathsf{S}(\underline{\mathsf{Z}})$, $evA(\mathsf{One}, \underline{\mathsf{S}(\mathsf{Z})}) = \mathsf{S}(\underline{\mathsf{S}(\mathsf{Z})})$, where we have underlined the hole position of the context. More generally, for a context $K_{\mathsf{One}} = \mathsf{S}(\bullet)$, we have $evA(\mathsf{One}, t) = K_{\mathsf{One}}[t]$ for any t. Thus, we can define a context-generating version evA_{c} of evA that satisfies $evA_{\mathrm{c}}(\mathsf{One}) = \mathsf{S}(\bullet)$, for example. The functions ev_{c} and evA_{c} can be defined as follows.

$$
\begin{array}{ll}
ev_{\mathrm{c}}(x) & = k[\mathsf{Z}] \ \textbf{where}\ k = evA_{\mathrm{c}}(x) \\
\\
evA_{\mathrm{c}}(\mathsf{One}) & = \mathsf{S}(\bullet) \\
evA_{\mathrm{c}}(\mathsf{Add}(x_1, x_2)) & = k_1[k_2[\bullet]] \ \textbf{where}\ \{k_i = evA_{\mathrm{c}}(x_i)\}_{i=1,2} \\
evA_{\mathrm{c}}(\mathsf{Dbl}(x)) & = k[k[\bullet]] \ \textbf{where}\ k = evA_{\mathrm{c}}(x)
\end{array}
$$

There are no accumulations or multiple data traversals. That is, evA_{c} is indeed a non-accumulative and input-linear *context-generating* transformation. Note that $ev_{\mathrm{c}}(x) = ev(x)$ holds for any x.

Now the simple inverse computation terminates again. For example, the inverse computation of ev_{c} for $\mathsf{S}^2(\mathsf{Z})$ is as follows.

$$
\begin{array}{ll}
 & (ev_{\mathrm{c}}(x) \stackrel{?}{=} \mathsf{S}^2(\mathsf{Z})) \\
\leadsto & \quad \{\text{because } (k[\mathsf{Z}] \stackrel{?}{=} \mathsf{S}^2(\mathsf{Z})) \equiv (k \stackrel{?}{=} \mathsf{S}^2(\bullet))\} \\
 & (evA_{\mathrm{c}}(x) \stackrel{?}{=} \mathsf{S}^2(\bullet)) \\
\leadsto_{x \mapsto \mathsf{Dbl}(x)} & \quad \{\text{because } (k[k[\bullet]] \stackrel{?}{=} \mathsf{S}^2(\bullet)) \equiv (k \stackrel{?}{=} \mathsf{S}[\bullet])\} \\
 & (evA_{\mathrm{c}}(x) \stackrel{?}{=} \mathsf{S}(\bullet)) \\
\leadsto_{x \mapsto \mathsf{One}} & \top
\end{array}
$$

The only difference is that now $\stackrel{?}{=}$ takes care of the contexts. Notice that the checks occurring in the narrowing have the form $f_{\mathrm{c}}(x) \stackrel{?}{=} K$, where K is a subcontext of the original output. Since this generally holds for ev_{c}, the termination property of the simple inverse computation is now recovered.

Besides the new point of view, our approach also involves a new way to express the memoized narrowing computation. Instead of using (a variant of) the existing method directly, we

use a tree automaton [10]; it is more suitable for a theoretical treatment than side-effectful memoized narrowing, and can express an infinite set of inputs (note that in general the number of corresponding inputs is infinite as in the case of *parity*). For example, the inverse computation of ev_{c} for $\mathsf{S}^2(\mathsf{Z})$ can be expressed by the following automaton where each state is of the form $q_{f^{-1}(K)}$.

$$
\begin{aligned}
q_{ev_{\mathrm{c}}^{-1}(\mathsf{S}^2(\mathsf{Z}))} &\leftarrow q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))} \\
q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))} &\leftarrow \mathsf{Add}(q_{evA_{\mathrm{c}}^{-1}(\bullet)}, q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))}) \\
q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))} &\leftarrow \mathsf{Add}(q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))}, q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))}) \\
q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))} &\leftarrow \mathsf{Add}(q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))}, q_{evA_{\mathrm{c}}^{-1}(\bullet)}) \\
q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))} &\leftarrow \mathsf{Dbl}(q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))}) \\
q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))} &\leftarrow \mathsf{One} \\
q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))} &\leftarrow \mathsf{Add}(q_{evA_{\mathrm{c}}^{-1}(\bullet)}, q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))}) \\
q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))} &\leftarrow \mathsf{Add}(q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))}, q_{evA_{\mathrm{c}}^{-1}(\bullet)}) \\
q_{evA_{\mathrm{c}}^{-1}(\bullet)} &\leftarrow \mathsf{Add}(q_{evA_{\mathrm{c}}^{-1}(\bullet)}, q_{evA_{\mathrm{c}}^{-1}(\bullet)}) \\
q_{evA_{\mathrm{c}}^{-1}(\bullet)} &\leftarrow \mathsf{Dbl}(q_{evA_{\mathrm{c}}^{-1}(\bullet)})
\end{aligned}
$$

Note that $f(x) \stackrel{?}{=} K$ can be regarded as $x \stackrel{?}{=} f^{-1}(K)$. We write $q_{f^{-1}(K)}$ for a state instead of $q_{f(x)\stackrel{?}{=}K}$ because an automaton constructed in this way can be regarded as all the possible reductions starting with $f^{-1}(K)$. This automaton contains the state $q_{evA_{\mathrm{c}}^{-1}(\bullet)}$ that accepts no trees, which intuitively means that the evaluation of $evA_{\mathrm{c}}^{-1}(\bullet)$ fails; *i.e.*, the narrowing from $evA_{\mathrm{c}}(x) \stackrel{?}{=} \bullet$ fails. The size of the resulting automaton is bounded linearly by the size of the original output of *ev*. It is also worth noting that we can extract a tree from an automaton in time linear to the size of the automaton [10].

All of the above results are obtained by just a simple observation: a program like *ev* is a non-accumulative context-generating transformation without multiple data traversals.

3 Target language

In this section, we formally describe the programs we target, which are written in an (untyped) first-order functional programming language with certain restrictions.

3.1 Values: trees

The values of the language are trees consisting of *constructors* (*i.e.*, a ranked alphabet).

Definition 1 (Trees) A set of *trees* $\mathcal{T}_\Sigma$ over constructors Σ is defined inductively as follows: for every $\sigma \in \Sigma^{(0)}$, $\sigma \in \mathcal{T}_\Sigma$, and for every $\sigma \in \Sigma^{(n)}$ and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$ $(n > 0)$, $\sigma(t_1, \ldots, t_n) \in \mathcal{T}_\Sigma$, where $\Sigma^{(n)}$ is the set of the constructors with arity n.

For constructors $\mathsf{Z}, \mathsf{Zero}, \mathsf{One}, \mathsf{Nil} \in \Sigma^{(0)}$, $\mathsf{S} \in \Sigma^{(1)}$ and $\mathsf{Cons}, \mathsf{Add} \in \Sigma^{(2)}$, examples of trees are $\mathsf{S}(\mathsf{Z})$, $\mathsf{Cons}(\mathsf{Z}, \mathsf{Nil})$, and $\mathsf{Add}(\mathsf{Add}(\mathsf{Zero}, \mathsf{One}), \mathsf{Zero})$. We shall fix the set Σ of the constructors throughout the paper for simplicity of presentation. The *size* of a tree t is the number of the constructor occurrences in t. For example, the size of $\mathsf{S}(\mathsf{Z})$ is 2.

In what follows, we shall use vector notation: $\overline{t}$ represents a sequence $t_1, \ldots, t_n$ and $|\overline{t}|$ denotes its length n.

Fig. 1 Syntax of the target language: σ is an n-ary constructor, and f is an $(m+1)$-ary function

$$
\begin{array}{lll}
\mathit{program} & ::= \mathit{rule}_1 \ldots \mathit{rule}_n & \\
\mathit{rule} & ::= f(p, y_1, \ldots, y_m) = e & \\
p & ::= x \mid \sigma(x_1, \ldots, x_n) & \\
e & ::= \sigma(e_1, \ldots, e_n) & \text{(Constructor Application)} \\
 & \mid \ f(x, e_1, \ldots, e_m) & \text{(Function Call)} \\
 & \mid \ y & \text{(Parameter Use)}
\end{array}
$$

3.2 Programs: macro tree transducers

The syntax of the language is shown in Fig. 1. A program consists of a set of rules, and each rule has the form of either $f(\sigma(x_1, \ldots, x_n), y_1, \ldots, y_m) = e$ or $f(x, y_1, \ldots, y_m) = e$. There are two kinds of variable: *input* and *output*. Input variables, denoted by x in Fig. 1, can be decomposed by pattern-matching but cannot be used to compose a result. Output variables, denoted by y in Fig. 1, can be used to compose a result but cannot be decomposed. Output variables are sometimes called (accumulation) parameters. A program in the language is nothing but a (stay) macro tree transducer (MTT) [15]. Thus, a program written in the target language is called an MTT in this paper.

Example 1 (*reverse*) A simple example of an accumulative function written in the target language is *reverse*. The following function *reverse* reverses a list of natural numbers expressed by S and Z.

$$
\begin{array}{ll}
\mathit{reverse}(x) & = \mathit{rev}(x, \mathsf{Nil}) \\
 & \\
\mathit{rev}(\mathsf{Nil}, y) & = y \\
\mathit{rev}(\mathsf{Cons}(a, x), y) & = \mathit{rev}(x, \mathsf{Cons}(\mathit{nat}(a), y)) \\
 & \\
\mathit{nat}(\mathsf{Z}) & = \mathsf{Z} \\
\mathit{nat}(\mathsf{S}(x)) & = \mathsf{S}(\mathit{nat}(x))
\end{array}
$$

The function *nat* just copies an input. This function is necessary because we prohibit using an input directly to produce a result in the language (see Fig. 1).

Example 2 (*eval*) The *eval* program in Sect. 1 is an example of an MTT program. So is its simplified version *ev*.

Example 3 (*mirror*) The following function *mirror* mirrors a list.

$$
\begin{array}{lll}
\mathit{mirror}(x) & = & \mathit{app}(x, \mathit{rev}(x, \mathsf{Nil})) \\
 & & \\
\mathit{app}(\mathsf{Nil}, y) & = & y \\
\mathit{app}(\mathsf{Cons}(a, x), y) & = & \mathsf{Cons}(\mathit{nat}(a), \mathit{app}(x, y))
\end{array}
$$

We omit the rules for *rev* and *nat* because they are the same as those in Example 1. Unlike *ev* and *eval*, *mirror* traverses an input twice with the different functions (*app* and *rev*). The function *app* is the so-called "append" function.

The *size* of a program is defined by the total number of function, constructor, and variable occurrences in the program. The intuition behind this definition is to approximate the size of program code in text. Note that the number of function or constructor occurrences is different from the number of functions or constructors. For example, the number of functions in *reverse* is 3, whereas the number of function occurrences is 9.

$$\frac{}{\Gamma;\, \Delta \uplus \{y \mapsto t\} \vdash y \downarrow t} \qquad \frac{\{\Gamma;\, \Delta \vdash e_i \downarrow t_i\}_{1 \le i \le n}}{\Gamma;\, \Delta \vdash \sigma(\overline{e}) \downarrow \sigma(\overline{t})}$$

$$\frac{f(x, \overline{y}) = e \quad \{\Gamma;\, \Delta \vdash e_i \downarrow t_i\}_{1 \le i \le |\overline{e}|} \quad \{x \mapsto s\};\, \left\{\overline{y} \mapsto \overline{t}\right\} \vdash e \downarrow t'}{\Gamma \uplus \{x \mapsto s\};\, \Delta \vdash f(x, \overline{e}) \downarrow t'}$$

$$\frac{f(\sigma(\overline{x}), \overline{y}) = e \quad s = \sigma(\overline{s}) \quad \{\Gamma;\, \Delta \vdash e_i \downarrow t_i\}_{1 \le i \le |\overline{e}|} \quad \{\overline{x} \mapsto \overline{s}\};\, \left\{\overline{y} \mapsto \overline{t}\right\} \vdash e \downarrow t'}{\Gamma \uplus \{x \mapsto s\};\, \Delta \vdash f(x, \overline{e}) \downarrow t'}$$

Fig. 2 Call-by-value semantics of the target language: here, we abuse the notation to write $\{\overline{x} \mapsto \overline{s}\}$ for $\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$ where $n = |\overline{x}| = |\overline{s}|$

The language has a standard call-by-value semantics, as shown in Fig. 2. A judgment $\Gamma;\, \Delta \vdash e \downarrow t$ means that under an input-variable environment Γ and output-variable environment Δ, an expression e is evaluated to a value t. Programs are assumed to be *deterministic*; *i.e.*, for each f, either f has at most one rule of the form $f(x, y_1, \ldots, y_m) = e$ or has at most one rule of the form $f(\sigma(x_1, \ldots, x_n), y_1, \ldots, y_m) = e$ for each σ. The semantics of a function f is defined by

$$[\![f]\!](s, \overline{t}) = \begin{cases} t' & \text{if } \{x \mapsto s\};\, \emptyset \vdash f(x, \overline{t}) \downarrow t' \text{ for fresh } x, \\ \bot & \text{otherwise.} \end{cases}$$

Note that we allow partial functions; *e.g.*, we have $[\![nat]\!](\mathsf{Nil}) = \bot$. We shall sometimes abuse the notation and simply write f for $[\![f]\!]$. The semantics is nothing but IO-production [15].

In addition, we also assume that programs are *nondeleting*, *i.e.*, every input variable must occur in the corresponding right-hand-side expression. This restriction does not change the expressiveness; we can convert any program to one satisfying this restriction by introducing the function *ignore* satisfying $[\![ignore]\!](s, t) = t$ for any s and t and defined by $ignore(\sigma(x_1, \ldots, x_n), y) = ignore(x_1, \ldots ignore(x_n, y) \ldots)$ for every $\sigma \in \Sigma$. The restriction simplifies the discussions in Sects. 4.3, 6.1 and 6.2. All the previous examples are deterministic and nondeleting.

A program is called *parameter-linear* if every output variable y occurring on the left-hand side occurs *exactly once* on the corresponding right-hand side of each rule.[5] All the previous examples are parameter-linear. Our polynomial time inverse computation depends on parameter-linearity.

4 Polynomial-time inverse computation

In this section, we formally describe our inverse computation. As briefly explained in Sect. 2, first, we convert an MTT program into a non-accumulative context-generating program (a program that generates contexts instead of trees) without multiple data traversals, such as ev_c in Sect. 2.3. Then, we perform inverse computation with memoization. More precisely, we construct a tree automaton [10] that represents the inverse computation result, whose run implicitly corresponds to (a context-aware version of) the existing inverse computation process with memoization [1, 4].

[5] Our definition of parameter-linearity is stronger than "single-use restricted on the parameters" [12] and "non-copying" [47]; they require that each parameter is used at most once.

Our inverse computation consists of three steps:

1. We convert a parameter-linear MTT into a non-accumulative context-generating program.
2. We apply tupling [8, 25] to eliminate multiple data traversals.
3. We construct a tree automaton that represents the inverse computation result.

The first two steps are to obtain a non-accumulative context-generating program without multiple data traversals. The third step represents memoized inverse computation. The rest of this section explains each step in detail.

4.1 Conversion to context-generating program

The first and most important step is to convert an MTT program into a non-accumulative context-generating program. This transformation is also useful for removing certain multiple data traversals, as shown in the example of *ev* in Sect. 2. Moreover, this makes it easy to apply tupling [8, 25] to programs. Note that viewing MTT programs as non-accumulative context-generating transformations is not a new idea (see Sect. 3.1 of [12] for example). The semantics of the context-generating programs shown later is nothing but using Lemma 3.4 of [12] to evaluate MTT programs.

First, we will give a formal definition of contexts.

Definition 2 An (m-hole) *context* K is a tree in $K \in \mathcal{T}_{\Sigma \cup \{\bullet_1, \ldots, \bullet_m\}}$ where $\bullet_1, \ldots, \bullet_m$ are nullary symbols such that $\bullet_1, \ldots, \bullet_m \notin \Sigma$.

An m-hole context K is *linear* if each $\bullet_i$ $(1 \leq i \leq m)$ occurs exactly once in K. We write $K[t_1, \ldots, t_m]$ for the tree obtained by replacing $\bullet_i$ with t_i for each $1 \leq i \leq m$. For example, $K = \mathsf{Cons}(\bullet_1, \bullet_2)$ is a 2-hole context and $K[\mathsf{Z}, \mathsf{Nil}]$ is the tree $\mathsf{Cons}(\mathsf{Z}, \mathsf{Nil})$. For 1-hole contexts, $\bullet_1$ is sometimes written as $\bullet$.

We showed that *ev* is indeed a non-accumulative context-generating transformation in Sect. 2. In general, any MTT program can be regarded as a non-accumulative context-generating transformation in the sense that, since output variables cannot be pattern-matched, the values bound to the output variables appear as-is in the computation result. Formally, we can state the following fact (Engelfriet and Vogler [15]; Lemma 3.19).

Fact 1 *$[\![f]\!](s, \overline{t}) = t$ if and only if there is K such that $[\![f]\!](s, \overline{\bullet}) = K$ and $t = K[\overline{t}]$.*

Accordingly, we can convert an MTT program into a non-accumulative context-generating program, as shown below.

Algorithm 1 Conversion to context-generating programs
Input: An MTT program
Output: A non-accumulative context-generating program
Procedure:
For each rule $f(p, y_1, \ldots, y_m) = e$ of the input program, construct a rule

$$f_{\mathrm{c}}(p) = e' \textbf{ where } k_{g_1, x_1} = g_{1\mathrm{c}}(x_1), \ldots, k_{g_n, x_n} = g_{n\mathrm{c}}(x_n)$$

where

– $g_{1\mathrm{c}}(x_1), \ldots, g_{n\mathrm{c}}(x_n)$ are obtained from all the function calls that occur as $g_i(x_i, \ldots)$ in e,

– k_{g_i,x_i} $(1 \le i \le n)$ represents a fresh variable name determined by g_i and x_i, and
– e' is obtained by $e' = \mathsf{cg}(e)$ where cg is defined as follows.

$$\begin{aligned}
\mathsf{cg}(\sigma(e_1, \ldots, e_n)) &= \sigma(\mathsf{cg}(e_1), \ldots, \mathsf{cg}(e_n)) \\
\mathsf{cg}(g(x, e_1, \ldots, e_n)) &= k_{g,x}[\mathsf{cg}(e_1), \ldots, \mathsf{cg}(e_n)] \\
\mathsf{cg}(y_j) &= \bullet_j \quad (1 \le j \le m)
\end{aligned}$$

As a result of the above, in a converted program, the arguments of every function are variables, and the return value of a function cannot be traversed again. This rules out any accumulative computation.

The algorithm above is very similar to that used for deaccumulation [19, 31]. Unlike deaccumulation, we treat contexts as first-class objects, which enables us to adopt special treatment for contexts in our inverse computation method.

Example 4 (*reverse*) The *reverse* program is converted into the following program.

$$\begin{aligned}
&\mathit{reverse}_\mathrm{c}(x) = k[\mathsf{Nil}] \ \textbf{where}\ k = \mathit{rev}_\mathrm{c}(x) \\
&\mathit{rev}_\mathrm{c}(\mathsf{Nil}) = \bullet_1 \\
&\mathit{rev}_\mathrm{c}(\mathsf{Cons}(a, x)) = k_2[\mathsf{Cons}(k_1, \bullet_1)] \\
&\quad \textbf{where}\ k_1 = \mathit{nat}_\mathrm{c}(a), k_2 = \mathit{rev}_\mathrm{c}(x) \\
&\mathit{nat}_\mathrm{c}(\mathsf{Z}) = \mathsf{Z} \\
&\mathit{nat}_\mathrm{c}(\mathsf{S}(x)) = \mathsf{S}(k) \ \textbf{where}\ k = \mathit{nat}_\mathrm{c}(x)
\end{aligned}$$

The converted program has no accumulative computation.

Example 5 (*eval*) The *eval* program in Sect. 1 is converted into the following program.

$$\begin{aligned}
&\mathit{eval}_\mathrm{c}(x) = k[\mathsf{Z}] \ \textbf{where}\ k = \mathit{evalA}_\mathrm{c}(x) \\
&\mathit{evalA}_\mathrm{c}(\mathsf{Zero}) = \bullet_1 \\
&\mathit{evalA}_\mathrm{c}(\mathsf{One}) = \mathsf{S}(\bullet_1) \\
&\mathit{evalA}_\mathrm{c}(\mathsf{Add}(x_1, x_2)) = k_1[k_2[\bullet_1]] \\
&\quad \textbf{where}\ k_1 = \mathit{evalA}_\mathrm{c}(x_1), k_2 = \mathit{evalA}_\mathrm{c}(x_2) \\
&\mathit{evalA}_\mathrm{c}(\mathsf{Dbl}(x)) = k[k[\bullet_1]] \ \textbf{where}\ k = \mathit{evalA}_\mathrm{c}(x)
\end{aligned}$$

Note that the two occurrences of the function call $\mathit{evalAcc}(x, \ldots)$ on the right-hand side of the rule $\mathit{evalAcc}(\mathsf{Dbl}(x)) = \ldots$ are unified into the single call $k = \mathit{evalAcc}_\mathrm{c}(x)$. Recall that Algorithm 1 generates a new variable $k_{f,x}$ for a pair of a function f and its input x, but not for its occurrence. Applying the same function to the same input results in the same context in a context-generating program, even though different accumulating arguments are passed in the original program. As a side effect, certain multiple data traversals, *i.e.*, traversals of the same input by the same function, are eliminated through this conversion.

Example 6 (*mirror*) The *mirror* program in Sect. 3 is converted into the following program.

$$\begin{aligned}
&\mathit{mirror}_\mathrm{c}(x) = k_1[k_2[\mathsf{Nil}]] \\
&\quad \textbf{where}\ k_1 = \mathit{app}_\mathrm{c}(x), k_2 = \mathit{rev}_\mathrm{c}(x) \\
&\mathit{app}_\mathrm{c}(\mathsf{Nil}) = \bullet_1 \\
&\mathit{app}_\mathrm{c}(\mathsf{Cons}(a, x)) = \mathsf{Cons}(k_1, k_2[\bullet_1]) \\
&\quad \textbf{where}\ k_1 = \mathit{nat}_\mathrm{c}(a), k_2 = \mathit{app}_\mathrm{c}(x)
\end{aligned}$$

Syntax

$$
\begin{array}{lll}
\mathit{prog} & ::= & \mathit{rule}_1 \ldots \mathit{rule}_n \\
\mathit{rule} & ::= & f(p) = e \ \textbf{where}\ k_1 = f_1(x_1), \ldots, k_n = f_n(x_n) \\
p & ::= & x \mid \sigma(x_1, \ldots, x_n) \\
e & ::= & \bullet_j \mid \sigma(e_1, \ldots, e_n) \mid k[e_1, \ldots, e_n]
\end{array}
$$

Semantics

$$
\frac{f(x) = e \ \textbf{where}\ \overline{k = g(z)} \quad l = |\overline{k = g(z)}| \quad \{\{x \mapsto s\} \vdash_{\mathrm{c}} g_i(z_i) \downarrow K_i\}_{1 \le i \le l} \quad \Theta = \{\overline{k} \mapsto \overline{K}\} \quad K' = e\Theta}{\Gamma \uplus \{x \mapsto s\} \vdash_{\mathrm{c}} f(x) \downarrow K'}
$$

$$
\frac{f(\sigma(\overline{x})) = e \ \textbf{where}\ \overline{k = g(z)} \quad s = \sigma(\overline{s}) \quad l = |\overline{k = g(z)}| \quad \{\{\overline{x} \mapsto \overline{s}\} \vdash_{\mathrm{c}} g_i(z_i) \downarrow K_i\}_{1 \le i \le l} \quad \Theta = \{\overline{k} \mapsto \overline{K}\} \quad K' = e\Theta}{\Gamma \uplus \{x \mapsto s\} \vdash_{\mathrm{c}} f(x) \downarrow K'}
$$

Fig. 3 Syntax and semantics of the converted programs: here, we abuse the notation to write $\overline{k = g(z)}$ for sequence $k_1 = g_1(z_1), \ldots, k_l = g_l(z_l)$ where $l = |\overline{k = g(z)}|$ and write $\{\overline{x} \mapsto \overline{s}\}$ as in Fig. 2

We have omitted the definitions of $\mathit{rev}_{\mathrm{c}}$ and $\mathit{nat}_{\mathrm{c}}$ because they are the same as in Example 4. Some multiple data traversals still remain as $k_1 = \mathit{app}_{\mathrm{c}}(x), k_2 = \mathit{rev}_{\mathrm{c}}(x)$. However, thanks to the conversion, this sort of multiple data traversal is easy to eliminate by tupling [8, 25] (see the next subsection).

For formal discussion, we define the syntax and the semantics of the non-accumulative context-generating programs in Fig. 3. Since contexts are bound to context variables k, the semantics uses *second-order substitutions* [12] that are mappings from variables to contexts. The application $e\Theta$ of a second-order substitution Θ to a term e is inductively defined by: $\sigma(e_1, \ldots, e_n)\Theta = \sigma(e_1\Theta, \ldots, e_n\Theta)$ and $k[e_1, \ldots, e_n]\Theta = K[e_1\Theta, \ldots, e_n\Theta]$ where $K = \Theta(k)$. Similarly to MTT, we write $[\![f]\!]$ for the semantics of f.

Now, we can show that the conversion is sound; it does not change the semantics of the functions.

Lemma 1 *For any tree* s, $[\![f]\!](s, \overline{\bullet}) = [\![f_{\mathrm{c}}]\!](s)$.

Together with Fact 1, we have $[\![f]\!](s, \overline{t}) = K[\overline{t}]$ with $K = [\![f_{\mathrm{c}}]\!](s)$ for every tree s and $\overline{t}$.

4.2 Tupling

Tupling is a well-known semantic-preserving program transformation that can remove some of the multiple data traversals [8, 25].

Roughly speaking, tupling transforms a rule

$$h(x) = \ldots k_1 \ldots k_2 \ldots \ \textbf{where}\ k_1 = f(x), k_2 = g(x)$$

into

$$h(x) = \ldots k_1 \ldots k_2 \ldots \ \textbf{where}\ (k_1, k_2) = \langle f, g \rangle(x).$$

Here, $\langle f, g \rangle$ is a function name introduced by tupling, and it is expected to satisfy $\langle f, g \rangle(x) = (f(x), g(x))$. Tupling tries to find a recursive definition of $\langle f, g \rangle(x)$ recursively.

For example, the following program for *mirror* is obtained by tupling.

$$\begin{aligned} &mirror_{\mathrm{c}}(x) && = k_1[k_2[\mathsf{Nil}]] \\ &\quad \textbf{where } (k_1, k_2) = \langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle(x) \end{aligned}$$

$$\begin{aligned} &\langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle(\mathsf{Nil}) && = (\bullet_1, \bullet_1) \\ &\langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle(\mathsf{Cons}(a, x)) && = (\mathsf{Cons}(k_1, k_2[\bullet_1]), k_3[\mathsf{Cons}(k_1, \bullet_1)]) \\ &\quad \textbf{where } k_1 = nat_{\mathrm{c}}(a), (k_2, k_3) = \langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle(x) \end{aligned}$$

We shall not explain the tupling in detail because it has been well-studied in the literature of functional programming [8, 25]. Moreover, we shall omit the formal definition of the syntax and the semantics of tupled programs because they are straightforward.

Note that we tuple only the functions that need to be tupled, *i.e.*, the functions that traverse the same input, for the sake of simplicity of our inverse computation method that we will discuss later. For example, app_{c} and rev_{c} are tupled because they traverse the same input, whereas nat_{c} and app_{c} are not tupled. Thus, the tupling step does not change the $reverse_{\mathrm{c}}$ and $eval_{\mathrm{c}}$ programs. In the tupled program obtained in this way, for any call of a tupled function $(k_1, \ldots, k_n) = \langle f_1, \ldots, f_n\rangle(x)$, each variable k_i $(1 \leq i \leq n)$ occurs at least once in the corresponding expression.

Thanks to the conversion described in the previous section, tupling can eliminate all the multiple data traversals from the converted programs. After tupling, a rule has the form of either

$$f(x) = \overline{e} \textbf{ where } \overline{k} = g(x)$$

or

$$f(\sigma(x_1, \ldots, x_n)) = \overline{e} \textbf{ where } \overline{k_1} = g_1(x_1), \ldots, \overline{k_n} = g_n(x_n).$$

Here, $f, g, g_1, \ldots, g_n$ are tupled functions. In other words, the tupled programs are always *input linear*; that is, every input variable occurring on the left-hand side also occurs *exactly once* on the corresponding right-hand side of each rule.

Tupling may cause size blow-up of a program: a tupled program is at worst 2^F-times as big as the original program; F here is the number of functions in the original program. Recall that we tuple only the functions that traverse the same input, not all the functions in a program. Note that only one of $\langle rev_{\mathrm{c}}, app_{\mathrm{c}}\rangle$ and $\langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle$ can appear in a tupled program. Thus, the tupled functions $\langle f_1, \ldots, f_n\rangle$ are as numerous as the sets of the original functions $\{f_1, \ldots, f_n\}$.

4.3 Tree automata construction as memoized inverse computation

We perform inverse computation with memoization after all the preprocessing steps have been completed. However, as mentioned in Sect. 2, unlike the existing inverse computation methods [1, 2, 4], we use a tree automaton [10] to express the memoized-inverse-computation result for the following reasons.

- A tree automaton is more suitable for a theoretical treatment than a side-effectful memoization table.
- The set $\{s \mid f(s) = t\}$ may be infinite (*e.g.*, *eval*).
- We can extract a tree (in DAG representation) from an automaton in time linear to the size of the automaton [10].
- In some applications such as test-case generation, it is more useful to enumerate the set of the corresponding inputs instead of returning one of the corresponding inputs.

Thus, the use of memoization is implicit in our inverse computation, and we shall not mention narrowing $\rightsquigarrow$ and check $\stackrel{?}{=}$ in this formal development. Note that tree automata are used in the inverse computation because they are *convenient* rather than *necessary*; even without them, we can use (a memoized and context-aware version of) the existing inverse computation methods [1, 2, 4].

First of all, we review the definition of tree automata. A *tree automaton* [10] $\mathcal{A}$ is a triple (Σ, Q, R), where Σ is a ranked alphabet, Q is a finite set of states, and R is a finite set of transition rules each having the form of either $q \leftarrow q'$ or $q \leftarrow \sigma(q_1, \ldots, q_n)$ where $\sigma \in Q^{(n)}$. We write $[\![q]\!]_{\mathcal{A}}$ for the trees accepted by state q in $\mathcal{A}$, *i.e.*, $\{t \mid q \leftarrow^* t\}$ where we take $\leftarrow$ as rewriting.

We shall roughly explain the construction of a tree automaton as inverse computation by using the example of ev_{c} given in Sect. 2. We construct an automaton whose states have the form $q_{f^{-1}(K)}$ that represents the evaluation of $f^{-1}(K)$, or the inverse computation result of f for K. Consider inverse computation of ev_{c} for $\mathsf{S}^2(\mathsf{Z})$. The idea behind the construction is to track the evaluation of $ev^{-1}(\mathsf{S}^2(\mathsf{Z}))$. Since the right-hand side of ev_{c} is $k[\mathsf{Z}]$, where $k = evA_{\mathrm{c}}(x)$, the evaluation $ev_{\mathrm{c}}^{-1}(\mathsf{S}^2(\mathsf{Z}))$ invokes the evaluation of $evA_{\mathrm{c}}^{-1}(k)$ for k such that $k[\mathsf{Z}] = \mathsf{S}^2[\mathsf{Z}]$. In this case, we have only such a $k = \mathsf{S}^2(\bullet)$. Thus, we generate a transition rule,

$$q_{ev_{\mathrm{c}}^{-1}(\mathsf{S}^2(\mathsf{Z}))} \leftarrow q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))}.$$

Next, let us focus on how $evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))$ is evaluated. There are three rules of evA_{c}. The first one has the right-hand side $\mathsf{S}(\bullet)$, the second one has the right-hand side $k_1[k_2[\bullet]]$ where $k_1 = evA_{\mathrm{c}}(x_1)$ and $k_2 = evA_{\mathrm{c}}(x_2)$, and the third one has the right-hand side $k[k[\bullet]]$ where $k = evA_{\mathrm{c}}(x)$. Then, we shall consider the (second-order) matching between the context $\mathsf{S}^2(\bullet)$, the argument of evA_{c}^{-1}, and the right-hand sides. The right-hand side of the first rule does not match the context. For the second rule, there are possibly three (second-order) substitutions obtained from matching $\mathsf{S}^2(\bullet)$ with $k_1[k_2[\bullet]]$: $k_1 = \bullet, k_2 = \mathsf{S}^2(\bullet)$; $k_1 = \mathsf{S}(\bullet), k_2 = \mathsf{S}(\bullet)$; and $k_1 = \mathsf{S}^2(\bullet), k_2 = \bullet$. Recall that k_1 and k_2 are defined by $k_1 = evA_{\mathrm{c}}(x_1)$ and $k_2 = evA_{\mathrm{c}}(x_2)$, and x_1 and x_2 come from the pattern $\mathsf{Add}(x_1, x_2)$. Thus, we generate the following rules.

$$\begin{aligned}
q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))} &\leftarrow \mathsf{Add}(q_{evA_{\mathrm{c}}^{-1}(\bullet)}, q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))}) \\
q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))} &\leftarrow \mathsf{Add}(q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))}, q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))}) \\
q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))} &\leftarrow \mathsf{Add}(q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))}, q_{evA_{\mathrm{c}}^{-1}(\bullet)})
\end{aligned}$$

Similarly, for the third rule, since there is only one substitution $k = \mathsf{S}(\bullet)$ obtained from matching $\mathsf{S}^2(\bullet)$ with $k[k[\bullet]]$, we generate the following rule.

$$q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))} \leftarrow \mathsf{Dbl}(q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))})$$

Now that we have obtained the transition rules corresponding to the call $evA_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))$, we focus on $evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))$. A similar discussion to the one above enables us to generate the following rules.

$$\begin{aligned}
q_{evA_{\mathrm{c}}(\mathsf{S}(\bullet))^{-1}} &\leftarrow \mathsf{One} \\
q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))} &\leftarrow \mathsf{Add}(q_{evA_{\mathrm{c}}^{-1}(\bullet)}, q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))}) \\
q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))} &\leftarrow \mathsf{Add}(q_{evA_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))}, q_{evA_{\mathrm{c}}^{-1}(\bullet)})
\end{aligned}$$

After that, we move to the rules of $evA_{\mathrm{c}}^{-1}(\bullet)$ and generate the following rules.

$$q_{evA_{\mathrm{c}}^{-1}(\bullet)} \leftarrow \mathsf{Add}(q_{evA_{\mathrm{c}}^{-1}(\bullet)}, q_{evA_{\mathrm{c}}^{-1}(\bullet)})$$
$$q_{evA_{\mathrm{c}}^{-1}(\bullet)} \leftarrow \mathsf{Dbl}(q_{evA_{\mathrm{c}}^{-1}(\bullet)})$$

Thus, the inverse computation of ev_{c} for $\mathsf{S}^2(\mathsf{Z})$ is complete. Let $\mathcal{A}_{\mathrm{I}}$ be the automaton constructed by gathering the generated rules. We can see that $[\![q_{ev_{\mathrm{c}}^{-1}(\mathsf{S}^2(\mathsf{Z}))}]\!]_{\mathcal{A}_{\mathrm{I}}} = \{\mathsf{Dbl}(\mathsf{One}), \mathsf{Add}(\mathsf{One}, \mathsf{One})\}$. Note that the state $q_{evA_{\mathrm{c}}^{-1}(\bullet)}$ accepts no trees.

This automaton construction is formalized as follows.

Algorithm 2

Input: A tupled program and a tree t.
Output: A tree automaton $\mathcal{A}_{\mathrm{I}} = (\Sigma, Q, R)$.
Procedure: Construct Q and R as follows.

- Q is the set of states of the form $q_{\langle f_1,\ldots,f_n\rangle^{-1}(K_1,\ldots,K_n)}$, where $\langle f_1, \ldots, f_n\rangle$ is a function occurring in the tupled program, K_i $(1 \leq i \leq n)$ is a $(a_i - 1)$-hole linear subcontext of t, and a_i is the arity of f_i. Here, K is called a subcontext of t if $t = K'[K[t_1, \ldots, t_m]]$ holds for some linear context K' and trees $t_1, \ldots, t_m$.
- R is the set of transition rules constructed from the rules of the tupled program and the tuples of the linear subcontexts of t, in the following way.
  - For each rule of the form $f(x) = \overline{e}$ **where** $\overline{k} = g(x)$ and subcontexts $\overline{K}$ of t, and for every second-order substitution Θ such that $\overline{e}\Theta = \overline{K}$, we construct a rule

$$q_{f^{-1}(\overline{K})} \leftarrow q_{g^{-1}(\overline{K'})}$$

    where $\overline{K'} = \overline{k}\Theta$.
  - For each rule of the form $f(\sigma(x_1, \ldots, x_n)) = \overline{e}$ **where** $\overline{k_1} = g_1(x_1), \ldots, \overline{k_n} = g_n(x_n)$ and contexts $\overline{K}$, and for every second-order substitution Θ such that $\overline{e}\Theta = \overline{K}$, we construct a rule

$$q_{f^{-1}(\overline{K})} \leftarrow \sigma(q_{g_1^{-1}(\overline{K'_1})}, \ldots, q_{g_n^{-1}(\overline{K'_n})})$$

    where $\overline{K'_i} = \overline{k_i}\Theta$ for each $1 \leq i \leq n$.

The problem of finding Θ satisfying $\overline{e}\Theta = \overline{K}$ for given $\overline{e}$ and $\overline{K}$ is called *second-order (pattern) matching*, and there have been proposed some algorithms to the problem [11, 26, 27]. In the actual construction of the automaton, we do not generate any state that cannot reach $q_{f^{-1}(t)}$, where f is the function to be inverted and t is the original output. The examples that will be discussed below use this optimization. Note that a tree is a 0-hole context. The nondeleting property is used in the above algorithm for simplicity. If a program is not nondeleting, some input variable x may not have the corresponding function call $g(x)$ in a rule of the tupled program. Then, we have to adopt special treatment for such a x in the construction of R in the algorithm.

Example 7 ($reverse_{\mathrm{c}}$) The following automaton $\mathcal{A}_{\mathrm{I}}$ is obtained from $reverse_{\mathrm{c}}$ and $t = \mathsf{Cons}(\mathsf{S}(\mathsf{Z}), \mathsf{Cons}(\mathsf{Z}, \mathsf{Nil}))$.

$$
\begin{array}{l}
q_{reverse_{\mathrm{c}}^{-1}(t)} \leftarrow q_{rev_{\mathrm{c}}^{-1}(\mathsf{Cons}(\mathsf{S}(\mathsf{Z}),\mathsf{Cons}(\mathsf{Z},\bullet_1)))} \\
\\
q_{rev_{\mathrm{c}}^{-1}(\mathsf{Cons}(\mathsf{S}(\mathsf{Z}),\mathsf{Cons}(\mathsf{Z},\bullet_1)))} \leftarrow \mathsf{Cons}(q_{nat_{\mathrm{c}}^{-1}(\mathsf{Z})}, q_{rev_{\mathrm{c}}^{-1}(\mathsf{Cons}(\mathsf{S}(\mathsf{Z}),\bullet_1))}) \\
q_{rev_{\mathrm{c}}^{-1}(\mathsf{Cons}(\mathsf{S}(\mathsf{Z}),\bullet_1))} \leftarrow \mathsf{Cons}(q_{nat_{\mathrm{c}}^{-1}(\mathsf{S}(\mathsf{Z}))}, q_{rev_{\mathrm{c}}^{-1}(\bullet_1)}) \\
q_{rev_{\mathrm{c}}^{-1}(\bullet_1)} \leftarrow \mathsf{Nil} \\
\\
q_{nat_{\mathrm{c}}^{-1}(\mathsf{S}(\mathsf{Z}))} \leftarrow \mathsf{S}(q_{nat_{\mathrm{c}}^{-1}(\mathsf{Z})}) \\
q_{nat_{\mathrm{c}}^{-1}(\mathsf{Z})} \leftarrow \mathsf{Z}
\end{array}
$$

We have $[\![q_{reverse_{\mathrm{c}}^{-1}(t)}]\!]_{\mathcal{A}_{\mathrm{I}}} = \{\mathsf{Cons}(\mathsf{Z}, \mathsf{Cons}(\mathsf{S}(\mathsf{Z}), \mathsf{Nil}))\}$, which means that there is only one input $s = \mathsf{Cons}(\mathsf{Z}, \mathsf{Cons}(\mathsf{S}(\mathsf{Z}), \mathsf{Nil}))$ satisfying $reverse(s) = reverse_{\mathrm{c}}(s) = t$.

Example 8 ($eval_{\mathrm{c}}$) The following automaton $\mathcal{A}_{\mathrm{I}}$, where q_i stands for state $q_{evalA_{\mathrm{c}}^{-1}(\mathsf{S}^i(\bullet_1))}$, is obtained from *eval* and $\mathsf{S}^2(\mathsf{Z})$.

$$
\begin{array}{l}
q_{eval_{\mathrm{c}}^{-1}(\mathsf{S}^2(\mathsf{Z}))} \leftarrow q_2 \\
\\
\begin{array}{lll}
q_2 \leftarrow \mathsf{Add}(q_2, q_0) & q_1 \leftarrow \mathsf{One} & q_0 \leftarrow \mathsf{Zero} \\
q_2 \leftarrow \mathsf{Add}(q_1, q_1) & q_1 \leftarrow \mathsf{Add}(q_1, q_0) & q_0 \leftarrow \mathsf{Add}(q_0, q_0) \\
q_2 \leftarrow \mathsf{Add}(q_0, q_2) & q_1 \leftarrow \mathsf{Add}(q_0, q_1) & q_0 \leftarrow \mathsf{Dbl}(q_0) \\
q_2 \leftarrow \mathsf{Dbl}(q_1) &&
\end{array}
\end{array}
$$

Intuitively, q_i represents the set of the arithmetic expressions that evaluate to $\mathsf{S}^i(\mathsf{Z})$.

Example 9 ($mirror_{\mathrm{c}}$) The following automaton $\mathcal{A}_{\mathrm{I}}$ is obtained from $mirror_{\mathrm{c}}$ and $\mathsf{Cons}(\mathsf{Z}, \mathsf{Cons}(\mathsf{Z}, \mathsf{Nil}))$.

$$
\begin{array}{l}
q_{mirror_{\mathrm{c}}^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Cons}(\mathsf{Z},\mathsf{Nil})))} \leftarrow q_{\langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Cons}(\mathsf{Z},\bullet_1)),\bullet_1)} \\
q_{mirror_{\mathrm{c}}^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Cons}(\mathsf{Z},\mathsf{Nil})))} \leftarrow q_{\langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle^{-1}(\mathsf{Cons}(\mathsf{Z},\bullet_1),\mathsf{Cons}(\mathsf{Z},\bullet_1))} \\
q_{mirror_{\mathrm{c}}^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Cons}(\mathsf{Z},\mathsf{Nil})))} \leftarrow q_{\langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle^{-1}(\bullet_1,\mathsf{Cons}(\mathsf{Z},\mathsf{Cons}(\mathsf{Z},\bullet_1)))} \\
\\
q_{\langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle^{-1}(\mathsf{Cons}(\mathsf{Z},\bullet_1),\mathsf{Cons}(\mathsf{Z},\bullet_1))} \leftarrow \mathsf{Cons}(q_{nat_{\mathrm{c}}^{-1}(\mathsf{Z})}, q_{\langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle^{-1}(\bullet_1,\bullet_1)}) \\
q_{\langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle^{-1}(\bullet_1,\bullet_1)} \leftarrow \mathsf{Nil} \\
\\
q_{nat_{\mathrm{c}}^{-1}(\mathsf{Z})} \leftarrow \mathsf{Z}
\end{array}
$$

We have $[\![q_{mirror_{\mathrm{c}}^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Cons}(\mathsf{Z},\mathsf{Nil})))}]\!]_{\mathcal{A}_{\mathrm{I}}} = \{\mathsf{Cons}(\mathsf{Z}, \mathsf{Nil})\}$. Note that some states occurring on the right-hand side do not occur on the left-hand side. An automaton with such states commonly appear when we try to construct an automaton for a function f and a tree t that is not in the range of f. For example, the following automaton $\mathcal{A}_{\mathrm{I}}$ is obtained from $mirror_{\mathrm{c}}$ and $\mathsf{Cons}(\mathsf{Z}, \mathsf{Nil})$.

$$
\begin{array}{l}
q_{mirror_{\mathrm{c}}^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Nil}))} \leftarrow q_{\langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle^{-1}(\mathsf{Cons}(\mathsf{Z},\bullet_1),\bullet_1)} \\
q_{mirror_{\mathrm{c}}^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Nil}))} \leftarrow q_{\langle app_{\mathrm{c}}, rev_{\mathrm{c}}\rangle^{-1}(\bullet_1,\mathsf{Cons}(\mathsf{Z},\bullet_1))}
\end{array}
$$

We have $[\![q_{mirror_{\mathrm{c}}^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Nil}))}]\!]_{\mathcal{A}_{\mathrm{I}}} = \emptyset$.

Our inverse computation is correct in the following sense.

Theorem 1 (Soundness and completeness) *For an input-linear tupled program,* $s \in [\![q_{\langle \overline{f} \rangle^{-1}(\overline{K})}]\!]_{\mathcal{A}_I}$ *if and only if* $[\![\langle \overline{f} \rangle]\!](s) = (\overline{K})$.

Proof Straightforward by induction. □

4.4 Complexity analysis of our inverse computation

We show that the inverse computation runs in time polynomial to the size of the original output and the size of the program, but in time exponential to the number of functions and the maximum arity of the functions and constructors. We state as such in the following theorem.

Theorem 2 *Given a parameter-linear MTT program that defines a function* f *and a tree* t*, we can construct an automaton representing the set* $\{s \mid f(s) = t\}$ *in time* $\mathrm{O}(2^F m(2^F n^{MF})^{N+1} n^{NMF})$ *where* F *is the number of the functions in the program,* n *is the size of* t*,* N *is the maximum arity of constructors in* Σ*,* m *is the size of the program, and* M *is the maximum arity of functions.*

Proof First, let us examine the cost of our preprocessing. The conversion into context-generating transformation does not increase the program size and can be done in time linear to the program size. In contrast, the tupling may increase the program size to $2^F m$. Thus, the total worst-case time complexity for preprocessing is $\mathrm{O}(2^F m)$.

Next, let us examine the cost of the inverse computation. The constructed automaton has at most $2^F n^{MF}$ states because every state is in the form $\langle g_1, \ldots, g_l \rangle^{-1}(K_1, \ldots, K_l)$, the number of $\langle g_1, \ldots, g_l \rangle$ is smaller than 2^F, the number of K_i is smaller than n^M, and l is no more than F. Note that the number of k-hole subcontexts in t is at most n^{k+1} and the contexts occurring in our inverse computation have at most $(M-1)$ kinds of holes. Since the number of the states in an automaton is bounded by $P = 2^F n^{MF}$ and the transition rules are obtained from the rules of the tupled programs that are smaller than $2^F m$, the number of the transition rules is bounded by $2^F m P^{N+1}$. Each rule construction takes $\mathrm{O}(n^{NMF})$ time because, for the second-order matching to find Θ such that $\overline{e}\Theta = \overline{K}$, the size of the solution space is bounded by $\mathrm{O}(n^{NMF})$; note that $\overline{e}$ contains at most NF context variables that have at most $(M-1)$ kind of holes. Thus, an upper bound of the worst-case cost of the inverse computation is $\mathrm{O}(2^F m(2^F n^{MF})^{N+1} n^{NMF})$.

Therefore, the total worst-case time complexity of our method is bounded by $\mathrm{O}(2^F m(2^F n^{MF})^{N+1} n^{NMF})$. □

Note that, if we start from input-linear tupled context-generating programs, the cost is $\mathrm{O}(m(Fn^{Md})^{N+1} n^{Mc})$, where c is the maximum number of context variables in the rules, and d is the maximum number of components of the tuples in the program. Also note that the above approximation is quite rough. For example, our method ideally runs in time linear to the size of the original output for *reverse* and *mirror* for *eval*, assuming some sophisticated second-order pattern matching algorithm under some sophisticated context representation depending on programs, which will be discussed in Sect. 5.5.

Each step of our inverse computation itself shown in Sects. 4.1, 4.2 and 4.3 does not use the parameter-linearity of an MTT. We only use the parameter-linearity to guarantee that our inverse computation is performed in polynomial time. For parameter-linear MTTs, we only have to consider linear contexts; the number of linear subcontexts of a tree t of size n is a polynomial of n, which leads our polynomial-time results. Our inverse computation

indeed terminates for MTTs without restrictions in exponential time because the number of possibly-non-linear m-hole subcontexts of a tree t is at most $|t|(m+1)^{|t|}$.

5 Experiments and discussions

In this section, we report our prototype implementation of the proposed algorithm and experimental results with the prototype system. The actual complexity of our inverse computation is unclear due to the two points: second-order matching and the automaton states actually generated by the automaton construction. By investigating several programs, we estimate the complexity of our method and clarify how these two points affect the computation cost.

After the experiments, we discuss how can we improve the complexity of our method for the investigated programs. For example, it is true that *ideally* we can achieve linear-time inverse computation for *reverse*, the linear-time inverse computation is hard to achieve with the naive implementation, as shown by the experimental result that we will describe later. We discuss what causes the gap and how we can remove the gap.

5.1 Implementation and environment

Our prototype system is written in Haskell, and is implemented as an inverse interpreter [1], *i.e.*, a program that takes a program and its output, and returns the corresponding inputs, rather than an inverse compiler (program inverter) [20]. Usually, inverse computation done by a inverse compiler runs faster than that done by an inverse interpreter. However, it is expected that the effect is rather small for our case which uses rather heavy computations, *i.e.*, the second-order pattern matching and the automaton construction. For the second-order matching, we used the algorithm in [44] without heuristics, which is a variant of Huet's algorithm [26] specialized to linear λ-terms.

The experiments below were carried out on Ubuntu Linux 12.04 (for i686) on a machine with Intel(R) Core(TM) i5 660 (3.33 GHz) and 8 GB memory. The prototype implementation is complied by Glasgow Haskell Compiler 7.4.1[6] under the flags `-O2 -rtsopts` and executed under the flags `+RTS -H`.

5.2 Experiments

To estimate how fast inverse computation can be performed by the prototype system in terms of the asymptotic complexity, we examined running time of the system by changing the size of original outputs fed to the inverse computation. The following programs and original outputs were tested.

- *reverse* in Example 1 and a list of Zs.
- *eval* in Sect. 1 and a natural number.
- *mirror* in Example 3 and a list of Zs.

[6] http://www.haskell.org/ghc/.

– The function *toc*, which construct the table-of-contents of a document and which will be discussed in Sect. 6.3, expressed as an MTT as below,

$$
\begin{aligned}
&toc(x) = \mathsf{UL}(mkToc(x, \mathsf{E}), \mathsf{E})\\
&mkToc(\mathsf{Title}(c, s), y) \quad = \mathsf{LI}(copy(c), mkToc(s, y))\\
&mkToc(\mathsf{Section}(c, s), y) \quad = \mathsf{LI}(\mathsf{UL}(mkToc(c, \mathsf{E}), \mathsf{E}), mkToc(s, y))\\
&mkToc(\mathsf{Paragraph}(c, s), y) = ignore(c, mkToc(s, y))\\
&copy(\mathsf{A}) = \mathsf{A}\\
&ignore(\mathsf{A}, y) = y
\end{aligned}
$$

and horizontally-repeated sequence representing (X)HTML fragments like:

```
<ul><li>A</li><li>A</li>...</ul>
```

Here, a fragment `<li>`x`</li>`y and `<ul>`x`</ul>`y are represented by $\mathsf{LI}(x, y)$ and $\mathsf{UL}(x, y)$ respectively, the text `A` is represented by A, and the empty sequence is represented by E.

– The program *toc* above and vertically-repeated (nested) (X)HTML fragments like

```
<ul><li>...<ul><li>A</li></ul>...</li></ul>
```

In the experiment, we only focus on estimation of the asymptotic complexity. For example, we do not focus on the overhead from manually-written inverse programs or the preprocessing cost.

Figure 4 shows the log-log plot of the experimental results. In a log-log plot, a function $y = cx^b$ is plotted as a straight line, because $y = cx^b$ implies $\log y = b \log x + \log c$. To show the estimated asymptotic complexity orders of the inverse computations, we also plotted an additional line in each plot. Note that these additional line are not obtained by fitting; they are added manually just to show how steeply the running time increases.

In the following, we discuss the experimental results one by one. Throughout the discussions, we use n for the size of the original output tree fed to the inverse computation that we focus on.

5.2.1 reverse

The running time of the inverse computation for *reverse* is estimated as $\mathrm{O}(n^2)$ from Fig. 4. One might think that this result is strange because we know that the inverse of *reverse* is *reverse* and thus can be executed in linear time. This gap comes from the two points: the construction of the tree automaton and the second-order pattern matching.

Regarding the construction of the automaton described in Sect. 4.3, we just used a pair $(\overline{f}, \overline{K})$ to represents a state $q_{\langle\overline{f}\rangle^{-1}(\overline{K})}$ in the automaton. Since in the construction we check if the transitions that go to a state are already generated or not and we used a balanced search tree[7] for the check, the checks takes $\mathrm{O}(|q| \log |q|)$ for $q = (\overline{f}, \overline{K})$. For *reverse*, since the constructed automaton contains $\mathrm{O}(n)$ states and each state has the size $\mathrm{O}(n)$, the construction itself takes $\mathrm{O}(n^2 \log n)$.

[7] Concretely, we used `Data.Map`.

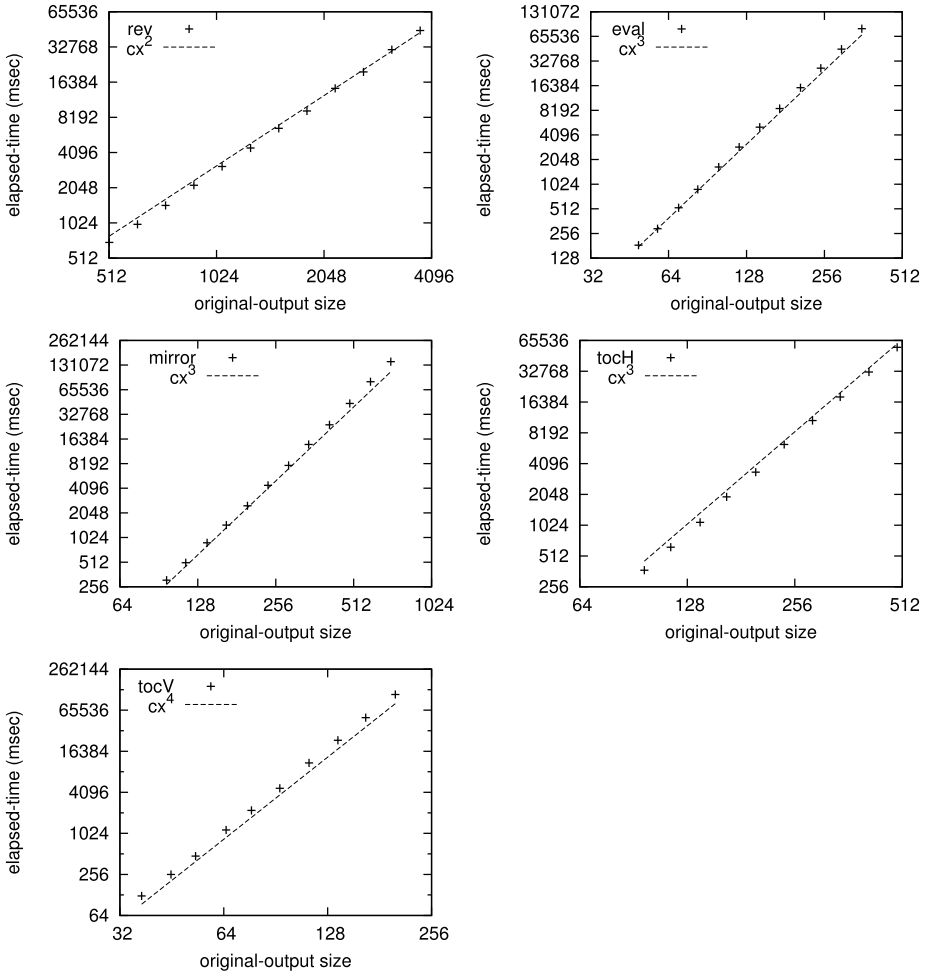

Fig. 4 Experimental results for *reverse*, *mirror*, *eval*, and *toc* for two kinds of outputs

In construction of the transitions of each state, we solve the following second-order matching problems.

$$k[\mathsf{Cons}(k', \mathsf{Nil})] \stackrel{?}{=} t \quad \text{and} \quad k[\mathsf{Cons}(k', \bullet)] \stackrel{?}{=} K$$

Here, we abuse the notation to use $\stackrel{?}{=}$ for the second-order matching problems. The implemented algorithm takes $O(n)$ time to find the solutions, because it searches $\mathsf{Cons}(\_, \mathsf{Nil})$/$\mathsf{Cons}(\_, \bullet)$ from the top of t/K. Thus, since we solve similar second-order matching problems for each state, the cost that comes from the second-order matching is $O(n^2)$.

One might notice that the experiment indicates that the time cost of inverse computation of $O(n^2)$ while the above discussion indicates that it is $O(n^2 \log n)$. Note that it is hard to observe the difference by $\log n$ because the factor is too small for the problem size. Thus, this is not a contradiction.

5.2.2 *mirror*

The running time of the inverse computation of *mirror* is estimated as $O(n^3)$ from Fig. 4. Note that the generated automaton contains $O(n^2)$ states because we do not know which part of the list is generated by *app* or *rev*. The constructed automaton contains states of the form $q_{\langle app,rev\rangle^{-1}(K^{l-k},K^{m-k})}$ where $l+m$ is equal to the length of an output list (thus, $l+m=(n-1)/2$) and $k \leq l, k \leq m$, and K^i denotes the context of the form of $\mathsf{Cons}(\mathsf{Z}, \mathsf{Cons}(\mathsf{Z}, \ldots, \mathsf{Cons}(\mathsf{Z}, \bullet)\ldots))$ containing i occurrences of Zs.

The effects of the automaton construction and the second-order matching are similar to those of *reverse*; the automaton construction and the second-order matching take $O(n \log n)$ and $O(n)$ time for each state, respectively.

5.3 *eval*

The inverse computation of *eval* is estimated to run in time $O(n^3)$ from Fig. 4. The constructed automaton contains $O(n)$ states and $O(n^2)$ transition rules.

In contrast to *reverse* and *mirror*, the implemented second-order pattern matching takes $O(n^2)$ time for each state. The matching $k[k[\bullet]] \stackrel{?}{=} K$ takes $O(|K|^2)$ time; the implemented algorithm guesses K_1 such that $K_1[K_2] = K$, in which there are $|K|$ candidates of such K_1, and then checks $K_1 = K_2$, which takes $O(|K|)$ time. The matching $k_1[k_2[\bullet]] \stackrel{?}{=} K$ also takes $O(|K|^2)$ time; the algorithm guesses K_1 such that $K_1[K_2] = K$ (similarly, there are $|K|$ candidates of such K_1) and takes $O(|K|)$ to check K_2 has the form $k_2[\bullet]$.

5.4 *toc*

From Fig. 4, the estimated complexity of the inverse computation of *toc* depends on what kind of trees we give to the inverse computation. The constructed automaton contains $O(n)$ states and $O(n^2)$ transitions for horizontally-repeated outputs, and contains $O(n^2)$ states and $O(n^3)$ transitions for vertically-repeated outputs. Note that the program of *toc* is converted to the following context-generating program.

$$
\begin{aligned}
&\mathit{toc}_{\mathrm{c}}(x) = \mathsf{UL}(k[\mathsf{E}], \mathsf{E}) \ \textbf{where}\ k = \mathit{mkToc}_{\mathrm{c}}(x)\\
&\mathit{mkToc}_{\mathrm{c}}(\mathsf{Title}(c,s)) \quad = \mathsf{LI}(k_1, k_2[\bullet])\\
&\qquad \textbf{where}\ k_1 = \mathit{copy}_{\mathrm{c}}(x), k_2 = \mathit{mkToc}_{\mathrm{c}}(s)\\
&\mathit{mkToc}_{\mathrm{c}}(\mathsf{Section}(c,s)) \quad = \mathsf{LI}(\mathsf{UL}(k_1[\mathsf{E}], \mathsf{E}), k_2[\bullet])\\
&\qquad \textbf{where}\ k_1 = \mathit{mkToc}_{\mathrm{c}}(c), k_2 = \mathit{mkToc}_{\mathrm{c}}(s)\\
&\mathit{mkToc}_{\mathrm{c}}(\mathsf{Paragraph}(c,s)) = k_1[k_2[\bullet]]\\
&\qquad \textbf{where}\ k_1 = \mathit{ignore}_{\mathrm{c}}(c), k_2 = \mathit{mkToc}_{\mathrm{c}}(s)\\
&\mathit{copy}_{\mathrm{c}}(\mathsf{A}) = \mathsf{A}\\
&\mathit{ignore}_{\mathrm{c}}(\mathsf{A}) = \bullet
\end{aligned}
$$

In horizontally-repeated outputs, which has the form $t = \mathsf{UL}(\mathsf{LI}(\mathsf{A}, \mathsf{LI}(\mathsf{A}, \ldots, \mathsf{LI}(\mathsf{A}, \mathsf{E})\ldots)), \mathsf{E})$, there are at most one k such that $\mathsf{UL}(k[\mathsf{E}], \mathsf{E}) = t$, while, in a vertically repeated outputs, which has the form $t = \mathsf{UL}(\mathsf{LI}(\ldots \mathsf{UL}(\mathsf{LI}(\mathsf{A}, \mathsf{E}), \mathsf{E})\ldots, \mathsf{E}), \mathsf{E})$, there are n candidates of k such that $\mathsf{UL}(k[\mathsf{E}], \mathsf{E}) = t$. This difference causes the difference in the number of states.

The second-order matching took $O(n^2)$ time for each state because we solve $k_1[k_2[\bullet]] \stackrel{?}{=} K$ which takes $O(|K|^2)$ time, similar to that in *eval*.

5.5 Discussions

In this section, we discuss how we can improve the asymptotic complexity of the implemented algorithm. Again, we use n for the size of an output tree that we focus on.

5.5.1 Pointer-representation of contexts

The prototype implementation uses very naive representation of contexts, *i.e.*, a tree with holes, which takes $O(n)$ space and the check of the equivalence also takes $O(n)$ time. Due to this cost, the running time of the inverse computation is usually no better than $O(n|Q|)$ where $|Q|$ is the number of states in the constructed automaton. For example, even for *nat* in Example 1, the inverse computation takes $O(n^2)$ time, in which the second-order matching $S(k) \stackrel{?}{=} t$—this is nothing but a first-order matching—can be solved in $O(1)$.

A possible solution to the problem is to represent a m-hole context by $(m+1)$ pointers (1 for its root and m for its holes). Since a pointer to the output tree t can be expressed in $O(\log n)$ space rather than n, it is expected that the representation reduces the cost of introduction of a state of a constructing automaton. Actually, this representation, combined with the "jumping" technique that will be discussed in Sect. 5.5.2, reduces the cost of the inverse computation for *nat*, *reverse*, while it may increase the cost in some cases as described later.

The procedure for the second-order pattern matching is changed according to the change of the representation of contexts. Note that we can see a pointer of an output tree as a state of an automaton representing the output tree. For example, a tree $\mathsf{S}(\mathsf{S}(\mathsf{Z}))$ can be expressed as an automaton below and each state $o\_$ of the automaton is essentially a pointer of the tree (here we subscript each state by the path from the root).

$$o_\varepsilon \leftarrow \mathsf{S}(o_1) \qquad o_1 \leftarrow \mathsf{S}(o_{11}) \qquad o_{11} \leftarrow \mathsf{Z}$$

Then, a context represented by pointers $o_0, o_1, \ldots, o_m$ where o_0 represents its root becomes a type $o_1 \to \cdots \to o_m \to o_0$, and the second-order matching problem $e \stackrel{?}{=} K$ becomes the typing problem—finding all the type environment Γ such that $\Gamma \vdash e :: \tau$ where τ is a pointer representation of K—in the intersection type system used by Kobayashi [29] with an additional restriction that Γ can have multiple entries of k like $k :: \tau_1, k :: \tau_2 \in \Gamma$ only if τ_1 and τ_2 represent the same context. Note that the intersection types do not appear explicitly here because we only consider linear contexts. For example, for $e = k_1[k_2[\mathsf{Z}]]$ and $\tau = o_\varepsilon$ in the automaton above, we find the three solutions $\Gamma = \{k_1 :: o_\varepsilon \to o_\varepsilon, k_2 :: o_{11} \to o_\varepsilon\}$, $\Gamma = \{k_1 :: o_1 \to o_\varepsilon, k_2 :: o_{11} \to o_1\}$ and $\Gamma = \{k_1 :: o_{11} \to o_\varepsilon, k_2 :: o_{11} \to o_{11}\}$, and for $e = k[k[\mathsf{Z}]]$ and the same τ, we find only one solution $\Gamma = \{k :: o_1 \to o_\varepsilon, k :: o_{11} \to o_1\}$, which denotes k is used twice to generate the original output tree where one is used to generate a context used to generate a tree at o_ε substituting a tree at o_1 to its hole and the other is used to generate a context used to generate a tree at o_1 substituting a tree at o_{11} to its hole, and these two contexts are the same. Again, the number of the solutions Γ is bounded by a polynomial of the number of the contexts, and thus a polynomial of n. Note that in the above example we can discard either one of $k :: o_1 \to o_\varepsilon$ and $k :: o_{11} \to o_1$ because they represent the same context.

With this pointer representation, the inverse computation for *nat* runs in time $O(n)$. To reduce the cost from $O(n)$ to $O(1)$, we have to know that *nat* is the identity function on natural numbers and is surjective, which is an orthogonal story to the discussions in this paper.

Note that there is a trade-off between this representation and the naive representation: a context can have n pointer-representations at worst. Thus, although this representation works effectively for *reverse*, *mirror* and *toc*, the constructed automaton for *eval* now has $O(n^2)$ states and $O(n^3)$ transitions in the pointer representation. Recall that it has $O(n)$ states and $O(n^2)$ transitions in the naive context representation. In the pointer representation, the same contexts $\mathsf{S}(\bullet)$ occurring in different positions are distinguished.

5.5.2 Jumping to arbitrary subtrees

As described above, the pointer representation is sometimes useful to reduce the cost of the automaton construction. However, to reduce the cost of the inverse computation, we have to reduce the cost of the second-order matching.

The pointer representation also sheds light on the problem, which enables us to traverse a tree or context from a leaf or arbitrary positions, while we have to traverse a tree or context from a top in the naive representation. For example, for *reverse* in which we solve the second-order matching problem $k_1[\mathsf{Cons}(k_2, \bullet)] \stackrel{?}{=} t$, we have to search $\mathsf{Cons}(k_2, \bullet)$ in t from the top in the naive representation. In contrast, in the pointer representation, the corresponding problem $\Gamma \vdash k_1[\mathsf{Cons}(k_2, \bullet)] :: o' \to o$ can be solved in constant time because we can "jump" to the hole position by searching transition rule $o'' \leftarrow \mathsf{Cons}(o''', o')$.

Finding a good strategy for typing would lead to an efficient second-order matching. Assuming some strategies such as performing "jumping" as possible, we can find that the inverse computation for *reverse* runs in time $O(n)$ and that for *mirror* runs in time $O(n^2)$. On the other hand, this technique does not reduce the inverse computation cost for *eval*.

5.5.3 Special treatment for monadic trees

More optimization can be applicable when the outputs are monadic trees, *i.e.*, trees built only from unary and nullary constructors such as $\mathsf{S}(\mathsf{S}(\mathsf{Z}))$ and $\mathsf{A}(\mathsf{B}(\mathsf{A}(\mathsf{E})))$. For monadic trees we can use integers for pointers.

Sometimes this integer-representation is useful to solve the second-order matching more efficiently. Consider the second-order matching $k[k[\bullet]] \stackrel{?}{=} \mathsf{S}^n(\bullet)$, which, in the integer-representation, can be translated to a problem that enumerating Γ such that $\Gamma \vdash k[k[\bullet]] :: n \to 0$ where n represents a pointer to the subtree occurring at depth n, or the subtree accessible from the root by a path with length n. Since the pattern is $k[k[\bullet]]$, we know that k splits the context $n \to 0$ in the middle. That is, n must be even and $\Gamma = \{k :: n/2 \to 0, k :: n \to n/2\}$. Thanks to the integer representation, we can find this unique candidate of k without investigating the context $\mathsf{S}^n(\bullet)$ at all; unlike the pointer representation, we can divide or multiply a "pointer" by a constant in the integer representation. Note that we still have to check if the types $n/2 \to 0$ and $n \to n/2$ represent the same context or not. The cost of the second-order matching is reduced from $O(n^2)$ to $O(n)$.

In general, for a second-order matching problem $e \stackrel{?}{=} K$, with the integer representation of contexts, we can represent constraints on "shape"s of the free variables in e by linear equations and inequalities. For example, from $\{k :: x_0 \to x_1, k :: x_2 \to x_3\} \vdash k[k[\bullet]] :: x_4 \to x_5$, we obtain a constraint on shape as $x_1 = x_5 \wedge x_2 = x_4 \wedge x_3 = x_0 \wedge (x_3 - x_2) = (x_1 - x_0) \wedge x_0 \leq x_1 \wedge x_2 \leq x_3 \wedge x_4 \leq x_5$. Solving the constraint for x_0, x_1, x_2, x_3, we get $x_1 = x_5, x_2 = x_4, 2x_0 = 2x_3 = x_4 + x_5$. By using the technique, the cost of the second-order matching in the inverse computation of *eval* becomes $O(n)$, and thus the cost of the inverse computation of *eval* becomes $O(n^3)$ again. This kind of technique is also applicable to list-generating programs like *reverse* and *mirror*, and functions of which outputs are partly monadic.

The similar optimization technique has been discussed also in the context of parsing of range concatenation grammars [7] in which users can represent arbitrary number of repetitions of a string. By using the technique, their parsing algorithm accepts 2^n repetitions of $\mathtt{a}$, *i.e.*, $\mathtt{a}^{2^n}$, in $O(2^n)$ time.

A much more specialized representation of contexts is applicable for natural numbers represented by S and Z, *i.e.*, trees built from one unary constructor and one nullary constructor. In this situation, we can represent a context by one natural number; for example, we can represents a context $\mathsf{S}(\mathsf{S}(\bullet))$ by 2. For *eval*, since there are no more redundant states in the representation and we can also apply the above optimization techniques to this representation, the inverse computation of *eval* runs in $O(n^2)$ time in this representation.

5.5.4 Estimation of shape

In the second-order matching, we have not used the fact that a variable k represents a return value of a function. Sometimes, we can perform for efficient inverse computation by using this information.

Consider the following rule of $mkToc_c$, which is one of the auxiliary functions used by toc_c.

$$mkToc_c(\mathsf{Paragraph}(c, s)) = k_1[k_2[\bullet]] \textbf{ where } k_1 = ignore_c(c), k_2 = mkToc_c(s)$$

According to this rule, the inverse computation of *toc* solves the second-order matching $k_1[k_2[\bullet]] \stackrel{?}{=} K$, which has as many solutions as the size of the context K. However, by using the fact that k_1 represents a return value of *ignore*, we know that k_1 must be $\bullet$. Combining the fact with other techniques described above, we can solve the second-order matching in constant time.

Exploiting this kind of information, we can achieve the linear time inverse computation for *mirror*. Recall that, even by using pointer representation, the inverse computation of *mirror* takes $O(n^2)$ time because the constructed tree automaton contains $O(n^2)$ states. This $O(n^2)$ numbers of states come form the rule

$$mirror_c(x) = k_1[k_2[\mathsf{Nil}]] \textbf{ where } (k_1, k_2) = \langle app_c, rev_c \rangle(x)$$

According to the rule, the inverse computation of *mirror* solves the second-order matching $k_1[k_2[\mathsf{Nil}]] \stackrel{?}{=} t$, which has as many solutions as the size of the output tree t, *i.e.*, $O(n)$ solutions. This is problematic because the inverse computation of each result takes $O(n)$ time, and the most of them produce no answers but introduce $O(n)$ states to the automaton. By using the information that *app* and *rev* generate the contexts of the same size, we can know that there is at most one solution for the matching, which makes the time complexity of the inverse computation of *mirror* be $O(n)$.

It would be a good future direction to discuss how can we estimate the shape and how can we use the estimated shape.

6 Extensions

We shall discuss four extensions of the inverse computation.

6.1 Pattern guards

Sometimes it is useful to define a function with pattern guards. For example, let us consider extending the simple arithmetic-expression language shown in Sect. 1 to include a conditional expression that branches by checking if a number is even or odd:

$$\textbf{data}\ \mathit{Exp} = \cdots \mid \mathsf{CaseParity}(\mathit{Exp}, \mathit{Exp}, \mathit{Exp})$$

According to the change, *eval* can also be naturally extended by using pattern guards:

$$\begin{aligned}
&\mathit{eval}(x) = \mathit{evalAcc}(x, \mathsf{Z})\\
&\quad\vdots\\
&\mathit{evalAcc}(\mathsf{CaseParity}(x, x_1, x_2), y) \mid \mathit{even}(x) = \mathit{evalAcc}(x_1, y)\\
&\mathit{evalAcc}(\mathsf{CaseParity}(x, x_1, x_2), y) \mid \mathit{odd}(x) \ = \mathit{evalAcc}(x_2, y)
\end{aligned}$$

Here, we have omitted the definition of *even*/*odd* that evaluates n and checks if the result is even/odd or not. We shall not discuss how they are defined at this point.

This extension can be achieved by using the known notion of MTT called *look-ahead* [15]. With regular look-ahead, we can test an input by using a tree automaton before we choose a rule. For example, *even* and *odd* can be seen as look-ahead because they can be expressed by the following tree automaton.

$$\begin{aligned}
&\mathit{even} \leftarrow \mathsf{Zero} && \mathit{even} \leftarrow \mathsf{CaseParity}(\mathit{even}, \mathit{even}, \mathit{even})\\
&\mathit{odd} \ \leftarrow \mathsf{One} && \mathit{even} \leftarrow \mathsf{CaseParity}(\mathit{even}, \mathit{even}, \mathit{odd})\\
&\mathit{even} \leftarrow \mathsf{Add}(\mathit{even}, \mathit{even}) && \mathit{even} \leftarrow \mathsf{CaseParity}(\mathit{odd}, \mathit{even}, \mathit{even})\\
&\mathit{even} \leftarrow \mathsf{Add}(\mathit{odd}, \mathit{odd}) && \mathit{even} \leftarrow \mathsf{CaseParity}(\mathit{odd}, \mathit{odd}, \mathit{even})\\
&\mathit{odd} \ \leftarrow \mathsf{Add}(\mathit{even}, \mathit{odd}) && \mathit{odd} \ \leftarrow \mathsf{CaseParity}(\mathit{even}, \mathit{odd}, \mathit{even})\\
&\mathit{odd} \ \leftarrow \mathsf{Add}(\mathit{odd}, \mathit{even}) && \mathit{odd} \ \leftarrow \mathsf{CaseParity}(\mathit{even}, \mathit{odd}, \mathit{odd})\\
&\mathit{even} \leftarrow \mathsf{Dbl}(\mathit{even}) && \mathit{odd} \ \leftarrow \mathsf{CaseParity}(\mathit{odd}, \mathit{even}, \mathit{odd})\\
&\mathit{even} \leftarrow \mathsf{Dbl}(\mathit{odd}) && \mathit{odd} \ \leftarrow \mathsf{CaseParity}(\mathit{odd}, \mathit{odd}, \mathit{odd})
\end{aligned}$$

Some pattern guards can be expressed by using regular look-ahead.

To handle regular look-ahead, we have to change the inverse computation method a bit. Consider a rule of the form,

$$f(x) \mid q(x) = g(x).$$

What transition rule should we produce from this f and a given K? Producing a rule $q_{f^{-1}(K)} \leftarrow q_{g^{-1}(K)}$ as the method discussed in Sect. 4.3 is unsatisfactory because the rule $f(x) \mid q(x) = g(x)$ is applicable only if x is accepted in q. Thus, we must embed the look-ahead information in the transition rule. This embedding can be naturally expressed by using an alternating tree automaton [10]:

$$q_{f^{-1}(K)} \leftarrow q_{g^{-1}(K)} \wedge q$$

However, using an alternating tree automaton does not fit our purpose because extracting a tree from an alternating tree automaton takes at worst time exponential to the size of the alternating tree automaton [10]; thus, it is difficult to bound the cost of our inverse computation polynomially to the original output size. Moreover, it also reduces the simplicity of the inverse computation method.

To keep our inverse computation method simple, we can specialize [35] the functions in a program to look-ahead as a preprocess. In a specialized program, for any function call $g(x, \overline{e})$ in a rule $f(p, \ldots) \mid \ldots q(x) \cdots = \ldots g(x, \overline{e}) \ldots$, the domain of the function must be accepted by the look-ahead; *i.e.*, $[\![g]\!](s, \overline{t}) = t$ implies $s \in [\![q]\!]$. Thus, in a specialized program, look-ahead cannot affect the inverse computation results (recall that programs are assumed to be nondeleting). For example, the specialized version of *evalAcc* is

$$
\begin{aligned}
&\mathit{evalAcc}(\mathsf{Zero}, y) = y\\
&\quad\vdots\\
&\mathit{evalAcc}(\mathsf{CaseParity}(x, x_1, x_2), y)\\
&\qquad \mid \mathit{even}(x) \quad = \mathit{ignore}_e(x, \mathit{ignore}(x_2, \mathit{evalAcc}(x_1, y)))\\
&\qquad \mid \mathit{odd}(x) \quad = \mathit{ignore}_o(x, \mathit{ignore}(x_1, \mathit{evalAcc}(x_2, y)))
\end{aligned}
$$

Recall that we use *ignore* because of the restriction that a program must use every input variable at least once. The functions ignore_e and ignore_o are specialized versions of *ignore* (to *even* and *odd* respectively):

$$
\begin{aligned}
&\mathit{ignore}_e(\mathsf{Zero}, y) = y\\
&\mathit{ignore}_e(\mathsf{Add}(x_1, x_2), y)\\
&\quad \mid \mathit{even}(x_1) \wedge \mathit{even}(x_2) = \mathit{ignore}_e(x_1, \mathit{ignore}_e(x_2, y))\\
&\quad \mid \mathit{odd}(x_1) \wedge \mathit{odd}(x_2) \quad = \mathit{ignore}_o(x_1, \mathit{ignore}_o(x_2, y))\\
&\mathit{ignore}_e(\mathsf{Dbl}(x), y) = \mathit{ignore}(x)\\
&\mathit{ignore}_e(\mathsf{CaseParity}(x, x_1, x_2), y)\\
&\quad \mid \mathit{even}(x) \wedge \mathit{even}(x_1) = \mathit{ignore}_e(x, \mathit{ignore}_e(x_1, \mathit{ignore}(x_2, y)))\\
&\quad \mid \mathit{odd}(x) \wedge \mathit{even}(x_2) \quad = \mathit{ignore}_o(x, \mathit{ignore}(x_1, \mathit{ignore}_e(x_2, y)))\\
&\mathit{ignore}_o(\mathsf{One}, y) = y\\
&\quad\vdots
\end{aligned}
$$

Here, we have omitted most of the definition of ignore_o. The above program has been simplified by using the fact that every input is either *even* or *odd*.

The specialization of a program increases the program size [33, 35]. In the worst case, a specialized program is $|Q|^N$ times as big as the original one and the specialization takes time proportional to the size of the specialized program, assuming that look-ahead is defined by a deterministic [10] tree automaton with the states Q, where N is the maximum arity of the constructors. Since this only increases the program size, our method still runs in time polynomial to the size of the original output.

6.2 Bounded use of parameters

The notion of look-ahead can relax the parameter-linearity restriction to finite-copying-in-parameter [12]. An MTT is called *finite-copying-in-parameter* [12] if there is a constant b such that K obtained by $[\![f]\!](s, \bullet_1, \ldots, \bullet_m) = K$ uses each hole $\bullet_j$ $(1 \leq j \leq m)$ at most b times for every function f of arity $m + 1$ and s. It is known that every finite-copying-in-parameter MTT can be converted into a parameter-linear MTT with look-ahead (see the proof of Lemma 6.3 in [12]). For example, the following MTT copies a parameter zero times or twice.

$$f(x) = g(x, \mathsf{A}) \quad g(\mathsf{A}, y) = \mathsf{C}(y, y) \quad g(\mathsf{B}, y) = \mathsf{D}$$

By using look-ahead, we can convert it into a parameter-linear MTT.

$$
\begin{array}{ll}
f(x) \mid q_2(x) = g_2(x, \mathsf{A}, \mathsf{A}) & \\
f(x) \mid q_0(x) = g_0(x) & \\
g_2(\mathsf{A}, y_1, y_2) = \mathsf{C}(y_1, y_2) & q_2 \leftarrow \mathsf{A} \\
g_0(\mathsf{B}) = \mathsf{D} & q_0 \leftarrow \mathsf{B}
\end{array}
$$

Here, g_i means g that copies the output variable i times and q_i means the set of the inputs for which g copies the output variable i times.

We can easily extend the method in Lemma 6.3 of [12] to generate specialized functions. A converted program can be $(b+1)^{MF(N+1)}$-times as big as the original one, where b is the bound of the parameter copies, N is the maximum arity of the constructors, F is the number of functions, and M is the maximum arity of the functions.

6.3 Parameter-linear macro forest transducers

A macro forest transducer [41], which is an important extension of a macro tree transducer, generates forests (roughly speaking, sequences of trees) instead of trees, which enables us to express XML transformations and serialization programs more directly. Our polynomial-time inverse computation results can be lifted to parameter-linear macro forest transducers.

Unlike macro tree transducers, in macro forest transducers, we can use the sequence concatenation "$\cdot$" in right-hand sides. The function *toc* defined below is an example of a macro forest transducer, which makes the table-of-contents of a document:

$$
\begin{array}{ll}
toc(x) = \mathsf{UL}(x) & \\
mkToc(\varepsilon) & = \varepsilon \\
mkToc(\mathsf{Title}(x_1) \cdot x_2) & = \mathsf{LI}(copy(x_1)) \cdot mkToc(x_2) \\
mkToc(\mathsf{Paragraph}(x_1) \cdot x_2) & = mkToc(x_2) \\
mkToc(\mathsf{Section}(x_1) \cdot x_2) & = \mathsf{LI}(\mathsf{UL}(mkToc(x_1))) \cdot mkToc(x_2)
\end{array}
$$

Here, ε denotes the empty sequence, and *copy* is defined as $copy(\sigma(x_1) \cdot x_2) = \sigma(copy(x_1)) \cdot copy(x_2)$ for any symbol σ and $copy(\varepsilon) = \varepsilon$. For an input $\mathsf{Title(A)} \cdot \mathsf{Section(Title(B)} \cdot \mathsf{Paragraph}(\ldots)) \cdot \mathsf{Title(C)}$, which represents an XML fragment

```
<title>A</title>
<section><title>B</title><paragraph>...</paragraph></section>
<title>C</title>
```

toc produces an output $\mathsf{UL(LI(A)} \cdot \mathsf{UL(LI(B))} \cdot \mathsf{LI(C))}$, which represents the XML fragment below.

```
<ul><li>A</li>
    <ul><li>B</li></ul>
    <li>C</li></ul>
```

In general, a rule of a (stay) macro forest transducer has the form of either

$$f(\varepsilon, \overline{y}) = e, \qquad f(\sigma(x_1) \cdot x_2, \overline{y}) = e, \quad \text{or} \quad f(x, \overline{y}) = e,$$

where each expression e is defined by the following BNF.

$$e ::= \sigma(e_1) \mid e_1 \cdot e_2 \mid y \mid f(x, e_1, \ldots, e_m)$$

A transformation like *toc*, which does not use accumulation parameters, can be expressed by a macro tree transducer, under some encoding of forests. However, this is not possible in general [41]; in general two macro tree transducers are required: one generates a forest in which the concatenations "·" are frozen [47] and the other thaws the frozen concatenations [41]. Thus, an extension is needed if we extend our results to parameter-linear macro forest transducers.

We also can perform polynomial time inverse computation for (deterministic) parameter-linear macro forest transducers. Recall that the following points are the keys to our polynomial-time result.

1. We can see a program as a linear non-accumulative context-generating program.
2. The number of contexts in a given output is bounded by a polynomial to the size of the output.
3. The substitutions of $\overline{e}\Theta = \overline{K}$ can be enumerated in polynomial time. Since the solution space is bounded polynomially by the second item, the existence of the polynomial-time checking of the equivalence of two contexts is sufficient.

Regarding Item 1, it is rather clear that we can apply the transformations discussed in Sect. 4.1 and Sect. 4.2 for macro forest transducers. Regarding Item 2, the number of m-hole linear contexts in a forest is bounded by $\mathrm{O}(n^{2m+2})$ where n is the size of the forest; there are $\mathrm{O}(n^2)$ subforests in a forest of size n similarly to the number of substrings in a string, and a linear m-hole context can be seen as a forest in which m subforests are replaced by holes. Item 3 is clear in the standard context representation in which a context is expressed as a forest with special symbols representing holes.

Note that, for linear macro forest transducers, where the uses of the both input and output variables are linear, polynomial-time inverse computation can be performed simply by preprocessing. For a linear macro forest transducer, the size of an output forest is bounded linearly by the size of the input forest, *i.e.*, the transformation is linear size increase [14]. Thus, a linear macro forest transducers is MSO-definable because it is expressed as compositions of MTTs and linear size increase [14]. Since MSO-definable tree transformation can be represented by a MTT that is both finite-copying-in-the-inputs and finite-copying-in-parameter [12], our method becomes applicable with some extra preprocessing as noted in Sect. 6.2.

6.4 Composing with inverse-image computation

Recall that our inverse computation method returns a set of corresponding inputs as a tree automaton. To enlarge the class of functions for which polynomial-time inverse computation can be performed, it is natural to try composing our inverse computation with inverse-image computation—computation of the set $\{s \mid f(s) \in T\}$ for a problem f and a given *set* of outputs T—which has been studied well in the context of tree transducers (for example, [15, 17, 32]).

However, there are several difficulties on this attempt:

– Usually, the inverse-image computation is harder than P. For example, it is known that the complexity of the inverse-image computation is EXPTIME-complete even for MTTs without output variables, which are thus non-accumulative, when T and the result set are given in tree automata [34].
– A few results are known on polynomial time inverse-image computation. However, some method [17] requires that a set of output trees must be given in a deterministic tree automaton; in general, converting a tree automaton to a deterministic one causes exponential size-blow-up.

– For some programs, polynomial-time inverse-image computation is possible even if we use a non-deterministic tree automaton to represents a set of outputs. However, composing these methods sometimes does not increase the expressive power. For example, although it is not difficult to see that polynomial-time inverse-image computation can be performed for MSO-definable transducers, the composition of a MSO-definable transducer followed by a parameter-linear MTT can also be expressed in a parameter-linear MTT [12].

We have to overcome the problems to enlarge the applicability of our method. Luckily, we can overcome the problems listed above.

– The inverse-image computation method proposed by Frisch and Hosoya [17] runs in polynomial-time for MTTs with the restriction of finite-input-copying-in-the-inputs [12].
– The method requires deterministic tree automata, but the automata obtained by our inverse computation can be converted to deterministic ones in polynomial time; there is no exponential size blow-up.
– The composition of a finite-input-copying-in-the-inputs MTT followed by a parameter-linear MTT can express a transformation that cannot be expressed in a single MTT, although the resulting class is artificial.

The following program *multiply*, which performs multiplication of two natural numbers, is an example for the third item.

$$multiply(x) = sum(dist(x))$$

$$\begin{aligned} dist(\mathsf{P}(x_1, x_2)) &= makeList(x_1, nat(x_2)) \\ makeList(\mathsf{Z}, y) &= \mathsf{Nil} \\ makeList(\mathsf{S}(x), y) &= \mathsf{Cons}(y, makeList(x, y)) \\ sum(\mathsf{Nil}) &= \mathsf{Z} \\ sum(\mathsf{Cons}(a, x)) &= add(a, sum(x)) \\ add(\mathsf{Z}, y) &= y \\ add(\mathsf{S}(x), y) &= \mathsf{S}(add(x, y)) \end{aligned}$$

Here, the function *nat* is the same as that in *reverse* and *mirror*. The function *multiply* defines a mapping $\mathsf{P}(\mathsf{S}^n(\mathsf{Z}), \mathsf{S}^m(\mathsf{Z})) \mapsto \mathsf{S}^{mn}(\mathsf{Z})$. Note that *multiply* is defined by a composition of the two functions: the one is *sum* written in a parameter-linear MTT, the other is *dist* written in a finite-input-copying-in-the-inputs MTT [12].

In the following, we show that the automata obtained by our inverse computation can be converted to deterministic ones in polynomial time. A tree automaton is called ϵ-free if it contains no rules of the form of $q \leftarrow q'$. A tree automaton is called *deterministic* [10] if it is ϵ-free and its transition rules contain no two different rules $q \leftarrow \sigma(q_1, \ldots, q_n)$ and $q' \leftarrow \sigma(q_1, \ldots, q_n)$ for any σ and $q_1, \ldots, q_n$. Note that we can convert a automaton to an ϵ-free one in polynomial-time [10].

A key property here is that, in an automaton obtained by our inverse computation, each state has the form $q_{\langle \overline{f} \rangle^{-1}(\overline{K})}$ and it satisfies that $s \in [\![q_{\langle \overline{f} \rangle^{-1}(\overline{K})}]\!]_{\mathcal{A}_\mathrm{I}}$ if and only if $[\![\langle \overline{f} \rangle]\!](s) = (\overline{K})$ (Theorem 1). Using the fact, we obtain the following lemma:

Lemma 2 *For any* $\overline{K}$, $\overline{K'}$, $\overline{f}$ *and* $\overline{f'}$ *satisfying* $K_i \neq K'_i$ *and* $f_i = f'_i$ *for some* i,

$$[\![q_{\langle \overline{f} \rangle^{-1}(\overline{K})}]\!]_{\mathcal{A}_\mathrm{I}} \cap [\![q_{\langle \overline{f'} \rangle^{-1}(\overline{K'})}]\!]_{\mathcal{A}_\mathrm{I}} = \emptyset$$

holds.

Proof We prove the lemma by contradiction. Suppose that we have $[\![q_{\overline{\langle f \rangle}^{-1}(\overline{K})}]\!]_{\mathcal{A}_\mathrm{I}} \cap [\![q_{\overline{\langle f' \rangle}^{-1}(\overline{K'})}]\!]_{\mathcal{A}_\mathrm{I}} \neq \emptyset$ for some $\overline{f}$, $\overline{f'}$, $\overline{K}$ and $\overline{K'}$ such that $f_i = f'_i$ and $K_i \neq K'_i$ for some i. According to Theorem 1, there exists some s such that $[\![\overline{\langle f \rangle}]\!](s) = \overline{K}$ and $[\![\overline{\langle f' \rangle}]\!](s) = \overline{K'}$. That is $[\![f_i]\!](s) = K_i$ and $[\![f'_i]\!](s) = K'_i$. Since $f_i = f'_i$ and $[\![f_i]\!]$ is a function because we consider deterministic MTTs, we have $K_i = K'_i$, which contradicts the assumption $K_i \neq K'_i$. □

The lemma guarantees that the naive subset-construction [10], which converts a tree automaton to a deterministic one, runs in polynomial-time. In the subset-construction, we construct an automaton whose states are represented by sets of states of the input automaton. A key property is that, in the constructed automaton, every state accepts at least one tree. Thus, if we apply the subset construction to the resulting automaton of our inverse computation, each state P in the generated automaton does not contain two states $q_{\overline{\langle f \rangle}^{-1}(\overline{K})}$ and $q_{\overline{\langle f' \rangle}^{-1}(\overline{K'})}$ with $K_i \neq K'_i$ and $f_i = f'_i$ in P. Therefore, the number of states in the constructed automaton is bounded by the number of mappings from a function f in the original program to a context K, which is $\mathrm{O}(n^{FM})$ where n is the size of an original output fed to the inverse computation, F is the number of functions in the original program and M is the maximum arity of the functions. Note that the number of states in the resulting automaton of our inverse computation is also $\mathrm{O}(n^{FM})$ (see the proof of Theorem 2).

Since we can convert an automaton obtained by our inverse computation to a deterministic one in polynomial-time, we can perform polynomial-time inverse computation for a transformation that is defined by a finite-copying-in-the-inputs MTT followed by a finite-copying-in-parameter MTT.

Note that the class of the transformations defined by a finite-copying-in-the-inputs MTT followed by a finite-copying-in-parameter MTT is incomparable with that defined by a MTT. In MTT, we can write a transformation that increases the size of a tree double-exponentially as below

$$\begin{aligned} &\mathit{dexp}(x) = \mathit{expBin}(x, \mathsf{L}) \\ &\mathit{expBin}(\mathsf{Z}, y) \quad = \mathsf{N}(y, y) \\ &\mathit{expBin}(\mathsf{S}(x), y) = \mathit{expBin}(x, \mathit{expBin}(x, y)) \end{aligned}$$

while we cannot express such a transformation in the class of the transformations defined by a finite-copying-in-the-inputs MTT followed by a finite-copying-in-parameter MTT. For a finite-copying-in-the-inputs MTT, it is not difficult to show that the height of an output tree is bounded linearly by the size of the input tree, and for a parameter-linear MTT (with look-ahead), it is known that the size of an output tree is bounded exponentially by the height of the input tree [47]. Thus, for the class of the composed transformations, the size of an output tree is bounded exponentially by the size of the input tree, which excludes *dexp*.

7 Related work

7.1 Inverse computation

There have been many studies on the inverse computation problem [1, 20, 21, 23, 30, 37, 40, 49]. They can be categorized into those on left-inverse computation and those on right-inverse computation. Left-inverse computation [20, 21, 23, 30, 40] focuses on injective functions and tries to make an efficient inverse computation based on injectivity analysis, but it can only handle provably-injective functions. Right-inverse computation [1, 37,

49] including ours can handle more functions than left-inverse computation does—it works even for non-injective functions—but the yielded inverse-computation process is usually much slower than that of left-inverse computation. Another important difference is that left-inverse computation is compositional; if we have effective left-inverse computation methods for f and g, we have an effective left-inverse computation method for $f \circ g$. On the other hand, right-inverse computation may not be compositional; even if we have right-inverse computation methods for f and g, then right-inverse computation may happen to be undecidable for $f \circ g$. Left-inverse computation is suitable for applications in which efficiency is the biggest concern, such as in serialization/deserialization. On the other hand, right-inverse computation is suitable for applications in which one wants to invert non-injective function to enumerate all the corresponding inputs, such as in test-case generation [9, 43]. It is worth noting that checking the injectivity of a function is generally undecidable. For parameter-linear MTTs in particular, the injectivity check is undecidable even if it has no output-variables [18] or it has no multiple data traversals (we can reduce the ambiguity check of a context-free grammar, which is known to be undecidable [24], to the problem). Thus, any left-inverse computation method essentially has a function written in parameter-linear MTT that cannot be inverted by it.

To the best of our knowledge, there are few discussions on the topic of multiple data traversals, except for Eppstein's work [16]. He demonstrated the usefulness of tupling [8, 25] that can make an injective function from non-injective functions.

Regarding accumulations, studies on left-inverse computation have treated them heuristically [20, 39, 40] because the injectivity check is usually undecidable with them. Glück and Kawabe [20] use the LR-parsing technique. In their system, if the grammar obtained from a program is LR-parsable, inverse program based on LR-parsing is derived. Note that their use of grammar is different from our use of tree automaton: their grammar represents a set of possible instruction sequences (traces) of a program while our tree automaton represents a set of inverse computation results. Nishida and Vidal [40] and Mogensen [39] focus on the special tail-recursive (thus usually accumulative) pattern and discuss the inverse computation of the pattern. Regarding right-inverse computation, although there are few studies focusing on accumulative functions, the approaches [17, 32] regarding the inverse-*image* computation have a strong connection to this work and will be discussed later in this section.

7.2 Results on tree transducers

We assumed that the programs are deterministic and showed that a tractable inverse computation is possible for parameter-linear MTTs. However, this result does not scale to nondeterministic programs. Even for MTTs without output variables, the problem of checking whether an inverse-computation result is empty or not is known to be NP-complete [42]. This means the complexity of the inverse computation problem of the nondeterministic MTTs even without output variables is NP-hard. For compositions of (deterministic/nondeterministic) macro tree transducers, checking whether an inverse-computation result is empty or not is known to be in NP [28]; thus the problem is NP-complete for compositions of nondeterministic macro tree transducers. It is still open whether the problem is NP-hard or not for compositions of deterministic macro tree transducers.

The problem of inverse computation takes a function f and an output tree t and returns the trees s such that $f(s) = t$. A similar problem, the inverse-image computation problem—computation of the set $\{s \mid f(s) \in T\}$ for a given f and T—has been studied on tree transducers (for example, [15, 17, 32]). The difference from the inverse computation

problem is that the inverse computation takes *one* tree but inverse-image computation takes *a set of* trees, and this difference is a key to our polynomial-time result. The complexity of the inverse-image computation is EXPTIME-complete even for the parameter-linear MTTs without output variables which are thus non-accumulative, when T and the result set are given in tree automata [34]. Roughly speaking, their EXPTIME-hard result is caused by intersections; for an expression like $\dots f(x) \dots f(x) \dots$ we essentially have to compute the intersection $\{s \mid f(s) \in T_1\} \cap \{s \mid f(s) \in T_2\}$ in the inverse-image computation [34]. On the other hand in our method, we do not need to compute the intersection because, for trees t_1 and t_2, $\{s \mid f(s) = t_1\} \cap \{s \mid f(s) \in t_2\}$ equals $\{s \mid f(s) = t_1\}$ if $t_1 = t_2$, and otherwise it is empty. This is implicitly expressed by the transformation in Sect. 4.1, in which we replace $\dots f(x) \dots f(x) \dots$ by $\dots k \dots k \dots$ **where** $k = f(x)$; a multiple data traversal is replaced by an output copying.

The observation that an MTT program is a non-accumulative context-generating transformation plays an important role in our method. A similar but different idea is exploited in inverse-image computation [17, 32]. Unlike our approach, the idea is to view an MTT program as a non-accumulative *mapping*-generating transformation, where a mapping is represented by input-output pairs. A context is different from a mapping; it contains more information than a mapping, *e.g.*, the information about the positions of holes. This difference results in the difference in inverse computation between our context-generation view and the mapping-generation view. The mapping-generation view considers mappings from a tuple of subtrees of t to a subtree of t for the original output t, which are indeed partially-applied functions such as $\lambda \overline{y}.[\![f]\!](s, \overline{y})$ used to generate t. However, the number of m-ary mappings on the subtrees of t is exponential to the size of t [17, 32]. Although the inverse computation based on the mapping-generation view can be performed in polynomial-time if there are no multiple data traversals [17], it is unclear whether polynomial-time inverse computation for functions with multiple-data traversals can be achieved or not. In contrast, we exploit the linearity of the holes—a context contains this information but a mapping does not—to achieve polynomial-time inverse computation for parameter-linear MTTs, in which a function can have multiple data traversals. Note that, like m-ary functions, the number of non-linear m-hole subcontexts in a tree is bounded exponentially by the size of the tree, whereas the number of linear ones is bounded polynomially by the size.

Regarding inverse computation of general MTTs, there is another polynomial-time inverse computation method besides ours that works for a subset of MTTs. The method of [17], as mentioned in the previous paragraph, runs in polynomial time for MTTs without multiple data traversals, *i.e.*, MTTs with the restriction of finite-input-copying-in-the-*inputs* [12]. In the restricted class of MTTs, we can copy an output unboundedly many times but we can traverse an input in only a bounded number of times. For example, *reverse* and *mirror* are finite-copying-in-the-inputs, but *eval* is not. In contrast, our method runs in polynomial time for (deterministic) MTTs with the restriction of finite-copying-in-the-*output* (Sect. 6.2), in which we can traverse an input unboundedly many times but we can copy an output only a bounded number of times. Whether we can perform polynomial-time inverse computation for general deterministic MTTs or not is still an open problem. It is worth noting that many useful functions can be written as an MTT in which both the input traversals and the output copies are bounded [12–14, 32], and thus inverse computation for the functions can be performed in polynomial time both by theirs and ours. Thus, the difference in expressiveness between ours and other methods is rather small, though not negligible. However, we claim that our method stands out by being systematic and simple.

7.3 (Formal) Grammars

In previous work [37], we have suggested an idea for inverse computation: We first find a grammar representing the possible outputs of a program, and then perform inverse computation through the parsing of the grammar. For left-inverse computation, we use an unambiguous grammar that over-approximates the possible outputs of a program, and for right-inverse computation, we use a grammar that exactly represents the possible outputs of a program but the grammar need not to be unambiguous. In [37], we have mainly discussed inverse computation using regular tree grammars [10], and showed that linear-time (right-)inverse computation is possible for affine and treeless [48] programs. For some subclass of MTTs, we can achieve polynomial-time inverse computation via parsing; for example, we can use context-free tree grammars for linear macro tree transducers and we can use IO macro grammars [5] for input-linear MTTs. However, to the best of our knowledge, there is no (tree) grammar that is powerful enough to express the possible outputs (range) of parameter-linear MTTs and is parsable in polynomial time.

However, this paper borrows the ideas from formal grammars in inverse computation. In this sense this work is also "grammar-based" as our previous work [37]. Concretely, we borrow the idea from the parsing algorithm of range concatenation grammars (RCG) [7]. RCG can mimic the ranges of some parameter-linear MTTs. Note that the following program *exp*, which is a simplified version of *ev* and *eval*, is a typical example of transformations that cannot be handled by the existing inverse computation methods

$$\begin{aligned} exp(x) &= ex(x, \mathsf{Z}) \\ ex(\mathsf{One}, y) &= \mathsf{S}(y) \\ ex(\mathsf{Dbl}(x), y) &= ex(x, ex(x, y)) \end{aligned}$$

We focus on RCG because it can express the possible outputs of *exp* and there is polynomial-time parsing algorithm. In RCG, we define a grammar by using Horn clauses as follows.

$$\begin{aligned} &\phi_{exp}(XY) \Leftarrow \phi_{exp}(X), \phi_{exp}(Y), eq(X, Y). \\ &\phi_{exp}(\mathtt{a}). \\ &eq(\mathtt{a}X, \mathtt{a}Y) \Leftarrow eq(X, Y). \\ &eq(\varepsilon, \varepsilon). \end{aligned}$$

This grammar expresses a set of strings $\{\mathtt{a}^{2^n} \mid n \in \mathbb{N}\}$. One might notice that the first rule of this grammar has a similar structure to the inverse computation for the rule of ex_c for Dbl

$$ex(\mathsf{Dbl}(x)) = k[k[\bullet]] \textbf{ where } k = ex(x)$$

where $ex^{-1}(K)$ try to find K' such that $K = K'[K'[\bullet]]$ and then try to compute $ex^{-1}(K')$. Only the difference is that RCG uses *eq* explicitly in some reason. (Note that this use of *eq* is mandatory in RCG because variables such as X and Y ranges over *occurrences* of strings (called *ranges* in [7]) instead of strings themselves, similar to the pointer-representation discussed in Sect. 5.5.1. But, this difference is ignorable if we do not use the same variable twice in the left-hand sides and do not allow concatenations in the right-hand sides.) In RCG, polynomial-time parsing is performed by range-wise memoization. We follow the idea and achieve the polynomial-time inverse computation by context-wise memoization.

One would notice that the idea of the parsing with range-wise memoization is already used Cocke-Younger-Kasami (CYK) parsing [24] for context-free grammars. Indeed, both

of the parsing algorithm of RCG and our inverse computation method are extensions of CYK parsing. We can see a context-free grammar as a transformation from a concrete syntax tree to the corresponding string, and we can write the transformation by a parameter-linear MTT. Then, our inverse computation for the transformation behaves as CYK parsing for the grammar, although the time complexity of the inverse computation depends on what second-order matching algorithm we use.

It is known that RCG is P-complete, *i.e.*, RCG can express any set of strings of which membership test is performed in polynomial-time [7]. However, we have not used the full-expressive power of RCG. Filling this gap is a future direction.

8 Conclusion

We have shown that viewing a function as a context-generating transformation simplifies inverse computation of accumulative functions with multiple data traversals. Accordingly, we can achieve systematic polynomial-time inverse computation with small modifications to the existing techniques.

A future direction is to develop a systematic program inversion method for accumulative functions based on the view point. Since now an accumulative function can be viewed as non-accumulative context-generating functions, we hope that we can extend usual range-analysis-based program-inversion methods [21, 30, 37] to those functions, and hope that a program-inversion method developed in this way would be a good alternative to the existing approaches [20, 39, 40]. Another future direction is to develop an inverse computation method that can handle more kinds of copying. One sort of the interesting copying in practice is those introduced by "join" operation in database query. Although this study is the first one to tackle the problem of "copies" in inverse computation, still there is a large gap between our results and the general "join" functions used in practice. Since tree transducers are hardly able to express "join"-like transformation [38], the next step in our research would be to identify what "join"s we should treat by designing an appropriate language.

Acknowledgements We wish to thank Akimasa Morihata, Meng Wang, and Soichiro Hidaka, who gave us many valuable comments on an earlier version of this work. The discussions in Sect. 5.5.1 are hinted from Naoki Kobayashi and Takeshi Tsukada. We also thank Janis Voigtländer who pointed out the relationship between Sect. 4.1 and deaccumulation. This work was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Research Activity Start-up 22800003, when the first author was in Tohoku University, and this work is partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Young Scientists (B) 24700020.

References

1. Abramov, S.M., Glück, R.: Principles of inverse computation and the universal resolving algorithm. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. Lecture Notes in Computer Science, vol. 2566, pp. 269–295. Springer, Berlin (2002)
2. Abramov, S.M., Glück, R., Klimov, Y.A.: An universal resolving algorithm for inverse computation of lazy languages. In: Virbitskaite and Voronkov [46], pp. 27–40
3. Albert, E., Vidal, G.: The narrowing-driven approach to functional logic program specialization. New Gener. Comput. **20**(1), 3–26 (2001)
4. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. J. ACM **47**(4), 776–822 (2000)

5. Asveld, P.R.: Time and space complexity of inside-out macro languages. Int. J. Comput. Math. **10**(1), 3–14 (1981)
6. Bird, R.: Introduction to Functional Programming Using Haskell, 2nd edn. Prentice Hall, New York (1998)
7. Boullier, P.: Range concatenation grammars. In: Bunt, H., Carroll, J., Satta, G. (eds.) New Developments in Parsing Technology. Text, Speech and Language Technology, vol. 23. Kluwer Academic, Dordrecht (2004). Chap. 13
8. Chin, W.-N., Khoo, S.-C., Jones, N.: Redundant call elimination via tupling. Fundam. Inform. **69**(1–2), 1–37 (2006)
9. Christiansen, J., Fischer, S.: EasyCheck—test data for free. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS. Lecture Notes in Computer Science, vol. 4989, pp. 322–336. Springer, Berlin (2008)
10. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007). http://www.grappa.univ-lille3.fr/tata
11. Dowek, G.: A second-order pattern matching algorithm for the cube of typed lambda-calculi. In: MFCS, pp. 151–160 (1991)
12. Engelfriet, J., Maneth, S.: Macro tree transducers, attribute grammars, and MSO definable tree translations. Inf. Comput. **154**(1), 34–91 (1999)
13. Engelfriet, J., Maneth, S.: A comparison of pebble tree transducers with macro tree transducers. Acta Inform. **39**(9), 613–698 (2003)
14. Engelfriet, J., Maneth, S.: Macro tree translations of linear size increase are MSO definable. SIAM J. Comput. **32**(4), 950–1006 (2003)
15. Engelfriet, J., Vogler, H.: Macro tree transducers. J. Comput. Syst. Sci. **31**(1), 71–146 (1985)
16. Eppstein, D.: A heuristic approach to program inversion. In: IJCAI, pp. 219–221 (1985)
17. Frisch, A., Hosoya, H.: Towards practical typechecking for macro tree transducers. In: Arenas, M., Schwartzbach, M.I. (eds.) DBPL. Lecture Notes in Computer Science, vol. 4797, pp. 246–260. Springer, Berlin (2007). Full version is available as Research Report, RR-6107, INRIA, 2007
18. Fülöp, Z.: Undecidable properties of deterministic top-down tree transducers. Theor. Comput. Sci. **134**(2), 311–328 (1994)
19. Giesl, J., Kühnemann, A., Voigtländer, J.: Deaccumulation techniques for improving provability. J. Log. Algebr. Program. **71**(2), 79–113 (2007)
20. Glück, R., Kawabe, M.: Derivation of deterministic inverse programs based on LR parsing. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS. Lecture Notes in Computer Science, vol. 2998, pp. 291–306. Springer, Berlin (2004)
21. Glück, R., Kawabe, M.: Revisiting an automatic program inverter for lisp. SIGPLAN Not. **40**(5), 8–17 (2005)
22. Glück, R., Sørensen, M.H.: Partial deduction and driving are equivalent. In: Hermenegildo, M.V., Penjam, J. (eds.) PLILP. Lecture Notes in Computer Science, vol. 844, pp. 165–181. Springer, Berlin (1994)
23. Gries, D.: The Science of Programming. Springer, Heidelberg (1981). Chap. 21 "Inverting programs"
24. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Prentice Hall, New York (2006). Chap. 7
25. Hu, Z., Iwasaki, H., Takeichi, M., Takano, A.: Tupling calculation eliminates multiple data traversals. In: ICFP, pp. 164–175 (1997)
26. Huet, G.P.: Résolution d'équations dans les langages d'ordre $1, 2, \ldots, \omega$. PhD thesis, Université de Paris VII (1976)
27. Huet, G.P., Lang, B.: Proving and applying program transformations expressed with second-order patterns. Acta Inform. **11**, 31–55 (1978)
28. Inaba, K., Maneth, S.: The complexity of tree transducer output languages. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS. LIPIcs, vol. 2, pp. 244–255. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik (2008)
29. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 416–428. ACM, New York (2009)
30. Korf, R.E.: Inversion of applicative programs. In: Hayes, P.J. (ed.) IJCAI, pp. 1007–1009. Kaufmann, Los Altos (1981)
31. Kühnemann, A., Glück, R., Kakehi, K.: Relating accumulative and non-accumulative functional programs. In: Middeldorp, A. (ed.) RTA. Lecture Notes in Computer Science, vol. 2051, pp. 154–168. Springer, Berlin (2001)
32. Maneth, S., Nakano, K.: XML type checking for macro tree transducers with holes. In: PLAN-X (2008)
33. Maneth, S., Perst, T., Seidl, H.: Exact XML type checking in polynomial time. In: Schwentick, T., Suciu, D. (eds.) ICDT. Lecture Notes in Computer Science, vol. 4353, pp. 254–268. Springer, Berlin (2007)
34. Martens, W., Neven, F.: On the complexity of typechecking top-down XML transformations. Theor. Comput. Sci. **336**(1), 153–180 (2005)

35. Matsuda, K., Hu, Z., Takeichi, M.: Type-based specialization of XML transformations. In: Puebla, G., Vidal, G. (eds.) PEPM, pp. 61–72. ACM, New York (2009)
36. Matsuda, K., Inaba, K., Nakano, K.: Polynomial-time inverse computation for accumulative functions with multiple data traversals. In: Kiselyov, O., Thompson, S. (eds.) PEPM, pp. 5–14. ACM, New York (2012)
37. Matsuda, K., Mu, S.-C., Hu, Z., Takeichi, M.: A grammar-based approach to invertible programs. In: Gordon, A.D. (ed.) ESOP. Lecture Notes in Computer Science, vol. 6012, pp. 448–467. Springer, Berlin (2010)
38. Milo, T., Suciu, D., Vianu, V.: Typechecking for XML transformers. J. Comput. Syst. Sci. **66**(1), 66–97 (2003)
39. Mogensen, T.Æ.: Report on an implementation of a semi-inverter. In: Virbitskaite and Voronkov [46], pp. 322–334
40. Nishida, N., Vidal, G.: Program inversion for tail recursive functions. In: Schmidt-Schauß, M. (ed.) RTA. LIPIcs, vol. 10, pp. 283–298. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik (2011)
41. Perst, T., Seidl, H.: Macro forest transducers. Inf. Process. Lett. **89**(3), 141–149 (2004)
42. Rounds, W.C.: Complexity of recognition in intermediate-level languages. In: FOCS, pp. 145–158. IEEE Press, New York (1973)
43. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and lazy SmallCheck: automatic exhaustive testing for small values. In: Gill, A. (ed.) Haskell, pp. 37–48. ACM, New York (2008)
44. Salvati, S., de Groote, P.: On the complexity of higher-order matching in the linear lambda-calculus. In: Nieuwenhuis, R. (ed.) RTA. Lecture Notes in Computer Science, vol. 2706, pp. 234–245. Springer, Berlin (2003)
45. Turchin, V.F.: The concept of a supercompiler. ACM Trans. Program. Lang. Syst. **8**(3), 292–325 (1986)
46. Virbitskaite, I., Voronkov, A. (eds.) Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI, Revised Papers, Novosibirsk, Russia, June 27–30, 2006. Lecture Notes in Computer Science, vol. 4378, Springer, Berlin (2007)
47. Voigtländer, J., Kühnemann, A.: Composition of functions with accumulating parameters. J. Funct. Program. **14**(3), 317–363 (2004)
48. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2), 231–248 (1990)
49. Yellin, D.M.: Attribute Grammar Inversion and Source-to-Source Translation. Lecture Notes in Computer Science, vol. 302. Springer, Berlin (1988)