Polynomial-Time Inverse Computation for Accumulative Functions with Multiple Data Traversals

Kazutaka Matsuda
Tohoku University
kztk@kb.ecei.tohoku.ac.jp

Kazuhiro Inaba
National Institute of Informatics[1]
kinaba@nii.ac.jp

Keisuke Nakano
The University of Electro-Communications
ksk@cs.uec.ac.jp

Abstract

Inverse computation has many 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.

Categories and Subject Descriptors I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program transformation; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Languages, Theory

Keywords Program Inversion, Inverse Computation, Program Transformation, Functional Programming, Tree Automata, Tree Transducers

1. Introduction

The problem of inverse computation [1, 15–17, 19, 24, 27, 32]—finding an input s for a program f and a given output t such that $f(s) = t$—has many 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 [4] even though we target an untyped first-order functional language with call-by-value semantics.)

$$\begin{array}{l}\mathbf{data}\ V = \mathsf{Z} \mid \mathsf{S}(V)\\ \mathbf{data}\ E = \mathsf{Zero} \mid \mathsf{One} \mid \mathsf{Add}(E,E) \mid \mathsf{Dbl}(E)\end{array}$$

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 $eval :: E \to V$ of the expressions can be implemented as follows.

$$\begin{array}{l}eval(x) = evalA(x, \mathsf{Z})\\ evalA(\mathsf{Zero}, y) = y\\ evalA(\mathsf{One}, y) = \mathsf{S}(y)\\ evalA(\mathsf{Add}(x_1, x_2), y) = evalA(x_1, evalA(x_2, y))\\ evalA(\mathsf{Dbl}(x), y) = evalA(x, evalA(x, y))\end{array}$$

Here, $eval$ uses $evalA$ that uses *accumulations* for efficiency. The function $evalA$ satisfies the invariant that $evalA(e, m) = eval(e) + m$, 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 $eval(\mathsf{Dbl}(x)) + y = eval(x) + eval(x) + y$.

The inverse computation of $eval$, which enumerates the inputs $\{s \mid eval(s) = t\}$ 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 $eval(e) = \mathsf{S}^{2^n}(\mathsf{Z})$ into $\mathsf{Dbl}^n(\mathsf{One})$, and we want to test if the optimizer works correctly or not, *i.e.*, whether $eval(e) = \mathsf{S}^{2^n}(\mathsf{Z})$ implies $f(e) = \mathsf{Dbl}^n(\mathsf{One})$ or not.[2] A solution would involve randomly generating or enumerating expressions e, filtering out the es that do not satisfy $eval(e) = \mathsf{S}^{2^n}(\mathsf{Z})$, 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). SmallCheck and EasyCheck use inverse computation for efficient test-case generation [6, 29], 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 $evalA$ contains *accumulations* and *multiple data traversals*. It is so far unclear how to perform tractable terminating inverse computation for func-

[1] He has moved to Google Inc. Email: kiki@kmonos.net

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

PEPM'12, January 23–24, 2012, Philadelphia, PA, USA.

tions with accumulations and multiple data traversals (Section 2). Some of the existing methods [1, 16, 24] do not terminate for functions with accumulations. Some approaches [15, 26, 27] 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 [13, 20], 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 [11] with the restriction of *parameter-linearity* (Section 3). In this class of functions, one cannot copy variables for accumulation (such as y in $evalA$) but one can traverse inputs (such as x, x_1, x_2 in $evalA$) many times. Our method computes the set $\{s \mid f(s) = t\}$ as a tree automaton [7] for a given function f and an output y in time polynomial to the size of y (Section 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, 3]. 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 shows two extensions of our proposal, and Section 6 shows the relationship between ours and the other research. Section 7 concludes the paper and outlines future work.

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, 3] terminates. The following function $parity$ takes a natural number n and returns $n \bmod 2$.

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

What should we do for inverse computation of $parity$ given an original output t? In [1], a symbolic computation method called (needed) narrowing [3] is used as a simple way to find a substitution θ such that $parity(x)\theta \stackrel{?}{=} t$, 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. Roughly speaking, a narrowing is a substitution followed by a reduction, and it can reduce an expression with free variables. For example, $parity(x)$ 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 $parity(x) \leadsto_{x \mapsto \mathsf{Z}} \mathsf{Z}$. The notion can naturally be extended to equivalence checks, such as $(parity(x) \stackrel{?}{=} \mathsf{Z}) \leadsto_{x \mapsto \mathsf{Z}} (\mathsf{Z} \stackrel{?}{=} \mathsf{Z}) \equiv \top$. 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[3]

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

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

$$\begin{array}{l} (parity(x) \stackrel{?}{=} \mathsf{Z}) \leadsto_{x \mapsto \mathsf{S}(x)} (aux(x) \stackrel{?}{=} \mathsf{Z}) \\ \leadsto_{x \mapsto \mathsf{S}(x)} (parity(x) \stackrel{?}{=} \mathsf{Z}) \leadsto_{x \mapsto \mathsf{Z}} \top \end{array}$$

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

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:

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

One might notice that the check $(parity(x) \stackrel{?}{=} \mathsf{S}^2(\mathsf{Z}))$ 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 $(parity(x) \stackrel{?}{=} \mathsf{S}^2(\mathsf{Z}))$ 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 the checks in the narrowing have the form of either $parity(x) \stackrel{?}{=} t$ or $aux(x) \stackrel{?}{=} t$, where t is the original output.

2.2 Problem: Non-Termination due to Accumulations and Multiple Data Traversals

Consider a simplified version of $eval$:

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

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.

Unlike $parity$, the simple inverse computation method does not always terminate. For example, the simple inverse computation of

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

ev for Z does not terminate.

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

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

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

  one can make the simple inverse computation terminate by decomposing $(evA(x_1, evA(x_2, \mathsf{Z})) \stackrel{?}{=} \mathsf{Z})$ 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 [13, 20]), 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 [13, 20] 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.

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

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 [7]; 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, 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 $q_{f^{-1}(K)}$.

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

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 [7].

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.

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

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

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

$$
\frac{\exists (f(x, \overline{y}) = e) \quad \{\Gamma; \Delta \vdash e_i \downarrow t_i\}_{1 \le i \le |\overline{e}|} \quad \{x \mapsto s\}; \{\overline{y} \mapsto \overline{t}\} \vdash e \downarrow v}{\Gamma \uplus \{x \mapsto s\}; \Delta \vdash f(x, \overline{e}) \downarrow v}
$$

$$
\frac{\exists (f(\sigma(\overline{x}), \overline{y}) = e).\ s = \sigma(\overline{s}) \quad \{\Gamma; \Delta \vdash e_i \downarrow t_i\}_{1 \le i \le |\overline{e}|} \quad \{\overline{x} \mapsto \overline{s}\}; \{\overline{y} \mapsto \overline{t}\} \vdash e \downarrow t}{\Gamma \uplus \{x \mapsto s\}; \Delta \vdash f(x, \overline{e}) \downarrow t}
$$

Figure 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 $n = |\overline{x}| = |\overline{s}|$.

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.

3.2 Programs: Macro Tree Transducers

The syntax of the language is shown in Figure 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 Figure 1, can be decomposed by pattern-matching but cannot be used to compose a result. Output variables, denoted by y in Figure 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) [11]. Thus, a program written in the target language is called an MTT in this paper.

Example 1. Simple example of an accumulative function written in the target language is $\mathit{reverse}$. The following function $\mathit{reverse}$ 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 Figure 1). □

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

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

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

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 $\mathit{reverse}$ is 3, whereas the number of function occurrences is 9.

The language has a standard call-by-value semantics, as shown in Figure 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 $[\![\mathit{nat}]\!](\mathsf{Nil}) = \bot$. We shall sometimes abuse the notation and simply write f for $[\![f]\!]$. The semantics is nothing but IO-production [11].

In addition, we also assume that 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 ign satisfying $[\![\mathit{ign}]\!](s, t) = t$ for any s and t and defined by $\mathit{ign}(\sigma(x_1, \ldots, x_n), y) = \mathit{ign}(x_1, \ldots \mathit{ign}(x_n, y) \ldots)$ for every $\sigma \in \Sigma$. All the previous examples satisfy these assumptions.

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.[4] 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 Section 2, first, we convert an MTT program into a non-accumulative context-generating program without multiple data traversals, such as ev_c in Section 2.3. Then, we perform inverse computation with memoization. More precisely, we construct a tree automaton [7] that represents the inverse computation result, whose run implicitly corresponds to (a context-aware version of) the existing inverse computation process with memoization [1, 3].

[4] Our definition of parameter-linearity is stronger than "single-use restricted on the parameters" [8] and "non-copying" [31]; 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 [5, 18] 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 Section 2. Moreover, this makes it easy to apply tupling [5, 18] to programs. Note that viewing MTT programs as non-accumulative context-generating transformations is not a new idea (see Section 3.1 of [8] for example). The semantics of the context-generating programs shown later is nothing but using Lemma 3.4 of [8] to evaluate MTT programs.

First, we will give a formal definition of contexts.

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

An n-hole context K is *linear* if each $\bullet_i$ $(1 \leq i \leq n)$ occurs exactly once in K. We write $K[t_1, \ldots, t_n]$ for the tree obtained by replacing $\bullet_i$ with t_i for each $1 \leq i \leq n$. 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 Section 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 [11]; 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

- $g_1(x_1), \ldots, g_n(x_n)$ are all the function calls that occur as $g_i(x_i, \ldots)$ in e,
- k_{g_i,x_i} $(1 \leq i \leq n)$ represents a fresh variable name determined by g_i and x_i, and
- e' is obtained from e by replacing each y_j $(1 \leq j \leq m)$ by $\bullet_j$ and replacing each call $g_i(x_i, \overline{e_i})$ $(1 \leq i \leq n)$ by $k_{g_i,x_i}[\overline{e'_i}]$, where $\overline{e'_i}$ are the results obtained by recursively applying the conversion to $\overline{e_i}$. □

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.

Example 4 ($reverse$)**.** The $reverse$ program can be converted into the following program.

$$\begin{aligned}
reverse_{\mathrm{c}}(x) &= k[\mathsf{Nil}] \textbf{ where } k = rev_{\mathrm{c}}(x)\\
rev_{\mathrm{c}}(\mathsf{Nil}) &= \bullet_1\\
rev_{\mathrm{c}}(\mathsf{Cons}(a, x)) &= k_2[\mathsf{Cons}(k_1, \bullet_1)]\\
&\quad \textbf{where } k_1 = nat_{\mathrm{c}}(a), k_2 = rev_{\mathrm{c}}(x)\\
nat_{\mathrm{c}}(\mathsf{Z}) &= \mathsf{Z}\\
nat_{\mathrm{c}}(\mathsf{S}(x)) &= \mathsf{S}(k) \textbf{ where } k = nat_{\mathrm{c}}(x)
\end{aligned}$$

The converted program has no accumulative computation. □

Example 5 ($eval$)**.** The $eval$ program in Section 1 can be converted into the following program.

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

Note that the two occurrences of the function call $evalA(x, \ldots)$ on the right-hand side of the rule $evalA(\mathsf{Dbl}(x)) = \ldots$ are unified into the single call $k = evalA_{\mathrm{c}}(x)$. 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 Section 3 can be converted into the following program.

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

We have omitted the definitions of rev_{c} and nat_{c} because they are the same as in Example 4. Some multiple data traversals still remain as $k_1 = app_{\mathrm{c}}(x), k_2 = rev_{\mathrm{c}}(x)$. However, thanks to the conversion, this sort of multiple data traversal is easy to eliminate by tupling [5, 18] (see the next subsection). □

For formal discussion, we define the syntax and the semantics of the non-accumulative context-generating programs in Figure 3. Since contexts are bound to context variables k, the semantics uses *second-order substitutions* [8] 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 [5, 18].

Syntax

$$
\begin{array}{lll}
prog & ::= & rule_1 \dots rule_n \\
rule & ::= & f(p) = e \ \mathbf{where}\ k_1 = f_1(x_1), \dots, k_n = f_n(x_n) \\
p & ::= & x \mid \sigma(x_1, \dots, x_n) \\
e & ::= & \bullet_j \mid \sigma(e_1, \dots, e_n) \mid k[e_1, \dots, e_n]
\end{array}
$$

Semantics

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

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

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

Roughly speaking, tupling transforms a rule

$$h(x) = \dots k_1 \dots k_2 \dots \ \mathbf{where}\ k_1 = f(x), k_2 = g(x)$$

into

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

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.

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

We shall not explain the tupling in detail because it has been well-studied in the literature of functional programming [5, 18]. 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 $(k_1, \dots, k_n) = \langle f_1, \dots, f_n \rangle(x)$, each variable k_i $(1 \le i \le n)$ 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

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

or

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

Here, $f, g, g_1, \dots, g_n$ 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 $\langle f_1, \dots, f_n \rangle$ are as numerous as the sets of the original functions $\{f_1, \dots, f_n\}$.

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 Section 2, unlike the existing inverse computation methods [1, 3], we use a tree automaton [7] 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 from an automaton in time linear to the size of the automaton [7].
- 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 $\leadsto$ 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, 3].

First of all, we review the definition of tree automata. A *tree automaton* [7] $\mathcal{A}$ 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 $q \leftarrow \sigma(q_1, \dots, q_n)$ 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 Section 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 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.

$$
\begin{array}{l}
q_{ev_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))} \leftarrow \mathsf{Add}(q_{ev_{\mathrm{c}}^{-1}(\bullet)}, q_{ev_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))}) \\
q_{ev_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))} \leftarrow \mathsf{Add}(q_{ev_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))}, q_{ev_{\mathrm{c}}^{-1}(\mathsf{S}(\bullet))}) \\
q_{ev_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))} \leftarrow \mathsf{Add}(q_{ev_{\mathrm{c}}^{-1}(\mathsf{S}^2(\bullet))}, q_{ev_{\mathrm{c}}^{-1}(\bullet)})
\end{array}
$$

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.

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

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

$$q_{evA_c(\mathsf{S}(\bullet))^{-1}} \leftarrow \mathsf{One}$$

After that, we move to the rules of $evA_c^{-1}(\bullet)$, but nothing is generated because no right-hand side matches with $\bullet$. 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 $[\![ev_c^{-1}(\mathsf{S}^2(\mathsf{Z}))]\!]_{\mathcal{A}_\mathrm{I}} = \{\mathsf{Dbl}(\mathsf{One}), \mathsf{Add}(\mathsf{One}, \mathsf{One})\}$.

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 $f(x) = \overline{e}\ \mathbf{where}\ \overline{k} = g(x)$ 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 $f(\sigma(x_1, \ldots, x_n)) = \overline{e}\ \mathbf{where}\ \overline{k_1} = g_1(x_1), \ldots, \overline{k_n} = g_n(x_n)$ 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 $1 \leq i \leq n$. □

Note that in the actual construction 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. Note that a tree is a 0-hole context.

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

$$\begin{aligned}
&q_{reverse_c^{-1}(t)} \leftarrow q_{rev_c^{-1}(\mathsf{Cons}(\mathsf{S}(\mathsf{Z}),\mathsf{Cons}(\mathsf{Z},\bullet_1)))}\\
&q_{rev_c^{-1}(\mathsf{Cons}(\mathsf{S}(\mathsf{Z}),\mathsf{Cons}(\mathsf{Z},\bullet_1)))}\\
&\qquad\leftarrow \mathsf{Cons}(q_{nat_c^{-1}(\mathsf{Z})}, q_{rev_c^{-1}(\mathsf{Cons}(\mathsf{S}(\mathsf{Z}),\bullet_1))})\\
&q_{rev_c^{-1}(\mathsf{Cons}(\mathsf{S}(\mathsf{Z}),\bullet_1))} \leftarrow \mathsf{Cons}(q_{nat_c^{-1}(\mathsf{S}(\mathsf{Z}))}, q_{rev_c^{-1}(\bullet_1)})\\
&q_{rev_c^{-1}(\bullet_1)} \leftarrow \mathsf{Nil}\\
&q_{nat_c^{-1}(\mathsf{S}(\mathsf{Z}))} \leftarrow \mathsf{S}(q_{nat_c^{-1}(\mathsf{Z})})\\
&q_{nat_c^{-1}(\mathsf{Z})} \leftarrow \mathsf{Z}
\end{aligned}$$

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

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

$$\begin{aligned}
&q_{eval_c^{-1}(\mathsf{S}^2(\mathsf{Z}))} \leftarrow q_2\\
&q_2 \leftarrow \mathsf{Add}(q_2, q_0) \quad q_1 \leftarrow \mathsf{One} \quad q_0 \leftarrow \mathsf{Zero}\\
&q_2 \leftarrow \mathsf{Add}(q_1, q_1) \quad q_1 \leftarrow \mathsf{Add}(q_1, q_0) \quad q_0 \leftarrow \mathsf{Add}(q_0, q_0)\\
&q_2 \leftarrow \mathsf{Add}(q_0, q_2) \quad q_1 \leftarrow \mathsf{Add}(q_0, q_1) \quad q_0 \leftarrow \mathsf{Dbl}(q_0)\\
&q_2 \leftarrow \mathsf{Dbl}(q_1)
\end{aligned}$$

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

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

$$\begin{aligned}
&q_{mirror_c^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Cons}(\mathsf{Z},\mathsf{Nil})))}\\
&\quad\leftarrow q_{\langle app_c, rev_c\rangle^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Cons}(\mathsf{Z},\bullet_1)),\bullet_1)}\\
&q_{mirror_c^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Cons}(\mathsf{Z},\mathsf{Nil})))}\\
&\quad\leftarrow q_{\langle app_c, rev_c\rangle^{-1}(\mathsf{Cons}(\mathsf{Z},\bullet_1),\mathsf{Cons}(\mathsf{Z},\bullet_1))}\\
&q_{mirror_c^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Cons}(\mathsf{Z},\mathsf{Nil})))}\\
&\quad\leftarrow q_{\langle app_c, rev_c\rangle^{-1}(\bullet_1,\mathsf{Cons}(\mathsf{Z},\mathsf{Cons}(\mathsf{Z},\bullet_1)))}\\
&q_{\langle app_c, rev_c\rangle^{-1}(\mathsf{Cons}(\mathsf{Z},\bullet_1),\mathsf{Cons}(\mathsf{Z},\bullet_1))}\\
&\quad\leftarrow \mathsf{Cons}(q_{nat_c^{-1}(\mathsf{Z})}, q_{\langle app_c, rev_c\rangle^{-1}(\bullet_1,\bullet_1)})\\
&q_{\langle app_c, rev_c\rangle^{-1}(\bullet_1,\bullet_1)} \leftarrow \mathsf{Nil}\\
&q_{nat_c^{-1}(\mathsf{Z})} \leftarrow \mathsf{Z}
\end{aligned}$$

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

$$\begin{aligned}
&q_{mirror_c^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Nil}))} \leftarrow q_{\langle app_c, rev_c\rangle^{-1}(\mathsf{Cons}(\mathsf{Z},\bullet_1),\bullet_1)}\\
&q_{mirror_c^{-1}(\mathsf{Cons}(\mathsf{Z},\mathsf{Nil}))} \leftarrow q_{\langle app_c, rev_c\rangle^{-1}(\bullet_1,\mathsf{Cons}(\mathsf{Z},\bullet_1))}
\end{aligned}$$

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

Our inverse computation is correct in the following sense.

Theorem 1 (Soundness and Completeness)**.** *For an input-linear tupled program, $s \in [\![q_{\langle\overline{f}\rangle^{-1}(\overline{K})}]\!]_{\mathcal{A}_\mathrm{I}}$ 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 $\mathrm{O}(2^F m(2^F n^{MF})^{N+1} n^c)$ where c is some constant, 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 $\mathrm{O}(n^c)$ time, where c is the maximum number of context compositions on the right-hand side, which intuitively represents the maximum degree of freedom in finding second-order substitutions. Thus, an upper bound of the worst-case cost of the inverse computation is $\mathrm{O}(2^F m (2^F n^{MF})^{N+1} n^c)$.

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

It is remarkable that if we start from input-linear tupled context-generating programs, the cost is $\mathrm{O}(m(Fn^{Md})^{N+1}n^c)$, where d is the maximum number of components of the tuples in the program fed to the inverse computation. Note that the above approximation is quite rough. For example, our method runs in time linear to the size of the original output for *reverse*, and runs in time quadratic to the size for *mirror* and *eval*.

5. Extensions

We shall discuss two extensions of the inverse computation.

5.1 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 Section 1 to include a conditional expression that branches by checking if a number is even or odd:

$$\mathbf{data}\ E = \ldots \mid \mathsf{CaseParity}(E, E, E)$$

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

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

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* [11]. 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.

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

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 Section 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 [7]:

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

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 [7]; 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 [23] 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 $f(p, \ldots) \mid \ldots q(x) \ldots = \ldots g(x, \overline{e}) \ldots$, 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. For example, the specialized version of *evalA* is

$$\begin{array}{l} \mathit{evalA}(\mathsf{Zero}, y) = y \\ \quad \vdots \\ \mathit{evalA}(\mathsf{CaseParity}(x, x_1, x_2), y) \\ \quad \mid \mathit{even}(x) \ = \mathit{ign}_e(x, \mathit{ign}(x_2, \mathit{evalA}(x_1, y))) \\ \quad \mid \mathit{odd}(x) \ = \mathit{ign}_o(x, \mathit{ign}(x_1, \mathit{evalA}(x_2, y))) \end{array}$$

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

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

Here, we have omitted most of the definition of ign_o.

The specialization of a program increases the program size [21, 23]. In the worst case, a specialized program is $|Q|^N$ times as big as the original one, assuming that look-ahead is defined by a deterministic [7] 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.

5.2 Bounded Use of Parameters

The notion of look-ahead can relax the parameter-linearity restriction to finite-copying-in-parameter [8]. An MTT is called *finite-copying-in-parameter* [8] 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 [8]). 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 [8] 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. Related Work

6.1 Inverse Computation

There have been many studies on the inverse computation problem [1, 15–17, 19, 24, 27, 32]. They can be categorized into ones on left-inverse computation and ones on right-inverse computation. Left-inverse computation [15–17, 19, 27] 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, 24, 32] 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 [6, 29]. 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 [14] or it has no multiple data traversals (we can reduce the emptiness check of the intersection of two context-free languages, which is known to be undecidable [2], 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 [12]. He demonstrated the usefulness of tupling [5, 18] that can make an injective function from non-injective functions.

Regarding accumulations, studies on left-inverse computation have treated them heuristically [15, 26, 27] because the injectivity check is usually undecidable with them. Glück and Kawabe [15] uses 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. Nishida and Vidal [27] and Mogensen [26] 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 [13, 20] regarding the inverse-*image* computation have a strong connection to this work and will be discussed later in this section.

6.2 Results on Tree Transducers and Formal Language

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 [28]. This means the complexity of the inverse computation problem of the nondeterministic MTTs even without output variables is NP-hard.

The problem of the 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, [11, 13, 20]). 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 [22]. Roughly speaking, their EXPTIME-hard result is caused by intersections; for an expression like $\ldots f(x) \ldots f(x) \ldots$ 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 [22]. 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 Section 4.1, in which we replace $\ldots f(x) \ldots f(x) \ldots$ by $\ldots k \ldots k \ldots$ **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 [13, 20]. Unlike ours, they 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 ours and theirs. From the mapping-generation view, they consider 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 [13, 20]. Although they can perform polynomial-time inverse computation if there are no multiple data traversals [13], it is unclear whether they can achieve polynomial-time inverse computation for functions with multiple-data traversals. 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 [13], 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* [8]. 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* (Section 5.2), 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 in MTT in which both the input traversals and the output copies are bounded [8–10, 20], 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. 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 [16, 19, 24] to those functions, and hope that a program-inversion method developed in this way would be a good alternative to the existing approaches [15, 26, 27]. 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 [25], 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. This work is partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Research Activity Start-up 22800003.

References

[1] S. M. Abramov and R. Glück. Principles of inverse computation and the universal resolving algorithm. In T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation*, *LNCS* 2566, pages 269–295. Springer, 2002.

[2] A. V. Aho and J. D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[3] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, 2000.

[4] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition edition, 1998.

[5] W.-N. Chin, S.-C. Khoo, and N. Jones. Redundant call elimination via tupling. *Fundam. Inform.*, 69(1-2):1–37, 2006.

[6] J. Christiansen and S. Fischer. EasyCheck — test data for free. In J. Garrigue and M. V. Hermenegildo, editors, *FLOPS*, *LNCS* 4989, pages 322–336. Springer, 2008.

[7] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007.

[8] J. Engelfriet and S. Maneth. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Inf. Comput.*, 154(1):34–91, 1999.

[9] J. Engelfriet and S. Maneth. Macro tree translations of linear size increase are MSO definable. *SIAM J. Comput.*, 32(4):950–1006, 2003.

[10] J. Engelfriet and S. Maneth. A comparison of pebble tree transducers with macro tree transducers. *Acta Inf.*, 39(9):613–698, 2003.

[11] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31(1):71–146, 1985.

[12] D. Eppstein. A heuristic approach to program inversion. In *IJCAI*, pages 219–221, 1985.

[13] A. Frisch and H. Hosoya. Towards practical typechecking for macro tree transducers. In M. Arenas and M. I. Schwartzbach, editors, *DBPL*, *LNCS* 4797, pages 246–260. Springer, 2007. Full version is available as Research Report, RR-6107, INRIA, 2007.

[14] Z. Fülöp. Undecidable properties of deterministic top-down tree transducers. *Theor. Comput. Sci.*, 134(2):311–328, 1994.

[15] R. Glück and M. Kawabe. Derivation of deterministic inverse programs based on lr parsing. In Y. Kameyama and P. J. Stuckey, editors, *FLOPS*, *LNCS* 2998, pages 291–306. Springer, 2004.

[16] R. Glück and M. Kawabe. Revisiting an automatic program inverter for lisp. *SIGPLAN Notices*, 40(5):8–17, 2005.

[17] D. Gries. *The Science of Programming*, chapter 21 Inverting Programs. Springer, Heidelberg, 1981.

[18] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *ICFP*, pages 164–175, 1997.

[19] R. E. Korf. Inversion of applicative programs. In P. J. Hayes, editor, *IJCAI*, pages 1007–1009. William Kaufmann, 1981.

[20] S. Maneth and K. Nakano. XML type checking for macro tree transducers with holes. In *PLAN-X*, 2008.

[21] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In T. Schwentick and D. Suciu, editors, *ICDT*, *LNCS* 4353, pages 254–268. Springer, 2007.

[22] W. Martens and F. Neven. On the complexity of typechecking top-down XML transformations. *Theor. Comput. Sci.*, 336(1):153–180, 2005.

[23] K. Matsuda, Z. Hu, and M. Takeichi. Type-based specialization of XML transformations. In G. Puebla and G. Vidal, editors, *PEPM*, pages 61–72. ACM, 2009.

[24] K. Matsuda, S.-C. Mu, Z. Hu, and M. Takeichi. A grammar-based approach to invertible programs. In A. D. Gordon, editor, *ESOP*, *LNCS* 6012, pages 448–467. Springer, 2010.

[25] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. Comput. Syst. Sci.*, 66(1):66–97, 2003.

[26] T. Æ. Mogensen. Report on an implementation of a semi-inverter. In Virbitskaite and Voronkov [30], pages 322–334.

[27] N. Nishida and G. Vidal. Program inversion for tail recursive functions. In M. Schmidt-Schauß, editor, *RTA*, volume 10 of *LIPIcs*, pages 283–298. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[28] W. C. Rounds. Complexity of recognition in intermediate-level languages. In *FOCS*, pages 145–158. IEEE, 1973.

[29] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and lazy SmallCheck: automatic exhaustive testing for small values. In A. Gill, editor, *Haskell*, pages 37–48. ACM, 2008.

[30] I. Virbitskaite and A. Voronkov, editors. *Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27-30, 2006. Revised Papers*, *LNCS* 4378, 2007. Springer.

[31] J. Voigtländer and A. Kühnemann. Composition of functions with accumulating parameters. *J. Funct. Program.*, 14(3):317–363, 2004.

[32] D. M. Yellin. *Attribute Grammar Inversion and Source-to-source Translation*, *LNCS* 302. Springer, 1988.