Noname manuscript No.
(will be inserted by the editor)

Functional Programs as Compressed Data

**Naoki Kobayashi · Kazutaka Matsuda ·
Ayumi Shinohara · Kazuya Yaguchi**

Received: date / Accepted: date

Abstract We propose an application of programming language techniques to lossless data compression, where tree data are compressed as functional programs that generate them. This "functional programs as compressed data" approach has several advantages. First, it follows from the standard argument of Kolmogorov complexity that the size of compressed data can be optimal up to an additive constant. Secondly, a compression algorithm is clean: it is just a sequence of β-expansions (i.e., the inverse of β-reductions) for λ-terms. Thirdly, one can use program verification and transformation techniques (higher-order model checking, in particular) to apply certain operations on data without decompression. In this article, we present algorithms for data compression and manipulation based on the approach, and prove their correctness. We also report preliminary experiments on prototype data compression/transformation systems.

Keywords Semantics based program manipulation · Program transformation · Data compression · Functional programs · Higher-order Model Checking

1 Introduction

Data compression plays an important role in today's information processing technologies. Its advantages are not limited to the decrease of data size, which enables more data to be stored in a device. Recent computer systems have a large memory hierarchy, from CPU registers to several levels of cache memory, main memory, hard disk,

This is an author's version of the paper. A definite version will appear in Higher-Order and Symbolic Computation. A preliminary summary of this article appeared in Proceedings of PEPM 2012, pp.121-130, 2012.

N. Kobayashi · K. Matsuda
Graduate School of Information Science and Technology, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo, 113-0033 Japan
E-mail: {koba,kztk}@is.s.u-tokyo.ac.jp

A. Shinohara · K. Yaguchi
Graduate School of Information Sciences, Tohoku University, 6-3-09 Aoba, Aramaki, Aoba-ku, Sendai, 980-8579 Japan
E-mail: {ayumi@,kazuya_yaguchi@shino.}ecei.tohoku.ac.jp

etc., so that decreasing the data size enables more data to be stored in faster memory, leading to more efficient computation. Some data compression schemes allow various operations to be performed without decompression in time polynomial in the size of the compressed data, so that one can sometimes achieve super-polynomial speed-up by compressing data. Data compression can also be applied to knowledge discovery [15].

In this paper, we are interested in the (lossless) compression of string/tree data as *functional programs*. The idea of "programs as compressed data" can be traced back at least to Kolmogorov complexity [28, 29], where the complexity of data is defined as the size of the smallest program that generates the data. The use of the λ-calculus in the context of Kolmogorov complexity has also been studied before [47]. Despite the generality and potential of the "functional programs as compressed data" (FPCD, for short) approach, however, it did not seem to have attracted enough attention, especially in the programming language community.

The goal of the present paper is to show that we can use programming language techniques, program verification/transformation techniques in particular, to strengthen the FPCD approach, so that the approach becomes not only of theoretical interest but potentially of practical interest. The approach has the following advantages.

1. Generality and optimality: In principle, it subsumes arbitrary compression schemes. Imagine some compression scheme and suppose that w is the compressed form of data v in the scheme. Let f be a functional program for decompression. Then, v can be expressed as the (closed) functional program $f\,w$. This is larger than w only by a constant, i.e. the size of the program f. This is actually the same as the argument for Kolmogorov complexity. We use a functional language (or more precisely, the λ-calculus) instead of a universal Turing machine, but it is easy to observe that the size of (a certain binary representation of) a λ-term representing the original data can be optimal with respect to Kolmogorov complexity, up to an additive constant.
We can also *naturally* mimic popular compression schemes used in practice. For example, consider the run-length coding. The string "abaabaabababbbb" can be compressed as [3, "aba", 4, "b"], meaning that the string consists of 3 repetitions of "aba" and 4 repetitions of "b". This can be expressed as:

```
(repeat 3 "aba" (repeat 4 "b" ""))
```

where `repeat` is a function that takes a non-negative integer n and strings s_1 and s_2, and returns the string $s_1^n s_2$. For another example, consider grammar-based compression, where strings or trees are expressed as (a restricted form of) context-free (tree) grammars [4, 18, 30]. The grammar-based compression has recently been studied actively, and used in practice for compression of XML data [4]. For instance, consider the tree shown in Figure 1 (which has been taken from [4]). It can be compressed as the following tree grammar:

$$S = B(B(A)) \qquad A = \mathtt{c}(\mathtt{a}, \mathtt{a}) \qquad B(y) = \mathtt{c}(A, \mathtt{d}(A, y))$$

Using the λ-calculus, we can express it by:

$$\textbf{let } A = \mathtt{c\,a\,a} \textbf{ in let } B = \lambda y.\mathtt{c}\,A\,(\mathtt{d}\,A\,y) \textbf{ in } B(B(A))$$

where the sharing of the tree context $\mathtt{c}(A, \mathtt{d}(A, [\,]))$ is naturally expressed by the λ-term. The data compression by a common pattern extraction then corresponds to an inverse β-reduction step. The previous grammar-based compression uses context-free grammars and their variants, while the λ-calculus has at least the same expressive power as higher-order grammars [9]. Thus, as far as data compression is concerned,

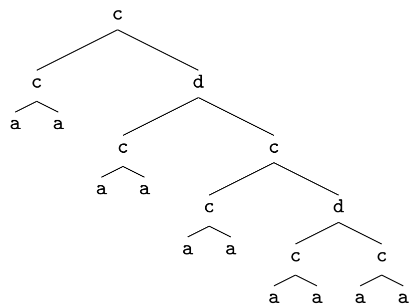

Fig. 1 A tree.

our approach can be considered a higher-order extension of the grammar-based compression. Our approach can achieve a theoretically higher compression ratio: the word generated by a straight-line program (a context-free grammar without recursion) can be only (single-)exponentially larger than the program, but the word generated by a λ-term can be hyperexponentially larger than the term. For example, as discussed in Example 1 in Section 2, the term:

$$(\lambda f.\underbrace{f\,f\,\cdots\,f}_{n}\,\mathtt{a}\,\mathtt{c})(\lambda g.\lambda x.g(g(x)))$$

generates the word $\mathtt{a}^{\overbrace{2^{2^{\cdots^{2}}}}^{n}}\mathtt{c}$. This is due to the use of higher-order functions.

2. Data manipulation without decompression: Besides the compression ratio and the efficiency of the compression/decompression algorithms, an important criterion is what operations can be directly applied to compressed data without decompression. In fact, the main strength of the grammar-based approach [4, 18, 31, 32, 37] is that a large set of operations, such as pattern matching and string replacement, can be performed without decompression. That is particularly important when the size of original data is too large to fit into memory, but the size of the compressed data is small enough. As we show in the present paper, the FPCD approach also enjoys such a property, by using program verification and transformation techniques. For example, consider a query q to ask whether a given tree T matches a certain pattern P. Given a program M as a compressed form of T, answering the query without decompressing M is considered a static program analysis problem: see Figure 2. If the pattern P is regular, then one can construct a corresponding tree automaton A_P that accepts the trees that match P. Thus, the problem can be further rephrased as: “Given a functional program M, is the tree generated by M accepted by A_P?” If M is a simply-typed program, then this is just an instance of higher-order model checking problems [21, 22, 34].
Pattern matching should often return not just a yes/no-answer, but extra information such as the position of the first match and the number of occurrences. Such operations can be expressed by tree transducers. Thus, the problem of performing such operations without decompression can be formalized as the following program transformation problem (see also the lower diagram in Figure 2):

> “Given a tree transducer f and a functional program M that generates a tree T, construct a program M' that generates $f(T)$.”

$$\begin{tikzcd}
\text{Tree data } T \arrow[r, "\text{query } q"] & \text{Yes/No} \\
\text{Program } M \arrow[u, "\begin{array}{l}\text{decompression}\\\text{(execution)}\end{array}"] \arrow[ur, "\text{program analysis}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
\text{Tree data } T \arrow[r, "\text{tree transducer } f"] & \text{Tree data } f(T) \\
\text{Program } M \arrow[u, "\begin{array}{l}\text{decompression}\\\text{(execution)}\end{array}"] \arrow[r, "\text{program transformation}"'] & \text{Program } M' \arrow[u, "\begin{array}{l}\text{decompression}\\\text{(execution)}\end{array}"']
\end{tikzcd}$$

Fig. 2 Query/transformation of compressed data as program analysis/transformation.

Thanks to the FPCD approach, the construction of the program M' is trivial: $M' = \hat{f}(M)$, where $\hat{f}$ is a representation of transducer f as a functional program. Of course, $\hat{f}(p)$ may not be an ideal representation, both in terms of the size of the program and the efficiency for further transformations. Fortunately, M is a tree generator and $\hat{f}$ is a consumer, so that we can apply the standard fusion transformation [13] to simplify $\hat{f}(M)$. An alternative, more sophisticated approach is, as discussed later, to extend a higher-order model checking algorithm to directly construct M'.

3. Applications to knowledge and program discovery: This is a more speculative advantage. It is folklore that compressed data contains the essence of the data, hence knowledge can be discovered by compressing data to the extreme [15]. As already discussed, the use of functional programs allows us to compress data to the limit (up to an additive term), so that we may be able to extract knowledge, represented in the form of a program, by compressing data. In fact, consider the following Church numeral representation of 9: $\lambda s.\lambda z.s(s(s(s(s(s(s(s(s(z)))))))))$. Our prototype compressor for λ-terms produces:

$$(\lambda n.\lambda f.n(nf))(\lambda s.\lambda x.s(s(s(x)))).$$

The part $\lambda s.\lambda x.s(s(s(x)))$ is the Church numeral 3, and the part $\lambda n.\lambda f.n(nf)$ is a square function for Church numerals (which is also the Church numeral 2). Thus, the equation $3^2 = 9$ and the square function have been automatically discovered by compression. Charikar et al. [5] notes "comprehensibility (to recognize patterns) is an important attraction of grammar-based compression relative to otherwise competitive compression schemes". As observed above, our FPCD approach has a similar advantage.

In the rest of this article, we first introduce the λ-calculus as the language for expressing compressed data, and discuss the relationship with Kolmogorov complexity in Section 2. We then describe an algorithm for compressing trees as λ-terms in Section 3. In Section 4, we extend and apply program verification/transformation techniques to achieve processing of compressed trees (represented in the form of λ-terms) without decompression. Section 5 reports preliminary experiments on data compression and processing. Section 6 discusses related work and Section 7 concludes.

The main contributions of this article are:

(i) Showing that *typed* λ-calculus with intersection types provides an optimal compression size up to an additive constant (Section 2.2).
(ii) Developing an algorithm to compress trees as λ-terms (Section 3).
(iii) Showing that higher-order model checking can be used to answer pattern match queries without decompression (Section 4.1).
(iv) An extension of higher-order model checking and an application of the fusion transformation to manipulate compressed data without decompression (Section 4.2).
(v) Implementation and experiments on the algorithms for data compression and data manipulations without decompression (Section 5).

The preliminary version of this article has appeared in [24]. From the previous version, we have added proofs, examples, discussions, and experiments.

2 λ-Calculus as a Data Compression Language

2.1 Syntax

We use the λ-calculus for describing tree data and tree-generating programs. To represent a tree, we assume a *ranked alphabet* (i.e., a mapping from a finite set of symbols to non-negative integers) Σ. We write $\mathtt{a}, \mathtt{b}, \dots$ for elements of the domain of Σ and call them *terminal symbols* (or just symbols). They are used as tree constructors below.

The set Terms_Σ of λ-*terms*, ranged over by M, is defined by:

$$M ::= x \mid a \mid \lambda x.M \mid M_1 M_2.$$

Here, the meta-variables x and a range over variables and symbols ($\mathtt{a}, \mathtt{b}, \dots$) respectively. Note that, if symbols are considered free variables, this is exactly the syntax of *the* λ-calculus. As usual, λx is a binder for the variable x, and we identify terms up to α-conversion. We also use the standard convention that the application $M_1 M_2$ is left-associative, and binds tighter than lambda-abstractions, so that $\lambda x.\mathtt{a}\,x\,x$ means $\lambda x.((\mathtt{a}\,x)\,x)$. We sometimes write $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ for $(\lambda x.M_2)M_1$. We write $\longrightarrow_\beta$ for the standard (one-step) β-reduction relation, and $\longrightarrow_\beta^*$ for its reflexive and transitive closure.

The *size* of M, written $\#M$, is defined by:

$$\#x = \#a = 1 \qquad \#(\lambda x.M) = \#M + 1 \qquad \#(M_1 M_2) = \#M_1 + \#M_2 + 1$$

The set of Σ-*labeled trees*, written $\mathcal{T}_\Sigma$, is the least subset of λ-terms closed under the rule:

$$\forall M_1, \dots, M_n \in \mathcal{T}_\Sigma. \Sigma(a) = n \Rightarrow a\ M_1\ \cdots\ M_n \in \mathcal{T}_\Sigma.$$

(Note here that n may be 0, which constitutes the base case.) We often use the meta-variable T to denote an element of $\mathcal{T}_\Sigma$.

If M has a β-normal form, we write $[\![M]\!]$ for it. In the present paper, we are interested in the case where $[\![M]\!]$ is a tree (i.e. an element of $\mathcal{T}_\Sigma$). When $[\![M]\!]$ is a tree T, we often call M a *program* that generates T, or a program for T in short. The goal of our data compression is, given a tree T, to find a small program for T.

Example 1 Let T be $\mathtt{a}^9(\mathtt{c})$, i.e., $\underbrace{\mathtt{a}(\mathtt{a}(\cdots(\mathtt{a}}_{9}(\mathtt{c}))\cdots))$.

It is generated by the following program M_1:

$$(\lambda n.n(n\,\mathtt{a})\mathtt{c})(\lambda s.\lambda x.s(s(s(x)))).$$

Note that $\#T = 19 > \#M_1 = 18$.

In general, the size of a program M for T can be hyper-exponentially smaller than the size of T. For example, consider the tree:

$$T := \mathtt{a}^{\overbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}^{n}}\,\mathtt{c}$$

It is generated by: $M_{2,n} := (\lambda f.\underbrace{f\,f\,\cdots\,f}_{n}\,\mathtt{a}\,\mathtt{c})(\lambda g.\lambda x.g(g(x)))$. □

Example 2 Consider the following term, which generates a unary tree $\mathtt{a}^{57}(\mathtt{c})$.

$$\begin{array}{l}
\textbf{let } b_0 = \lambda n.\lambda s.\lambda z.n\,s\,(n\,s\,z) \textbf{ in}\\
\textbf{let } b_1 = \lambda n.\lambda s.\lambda z.s\,(n\,s\,(n\,s\,z)) \textbf{ in}\\
\textbf{let } \mathit{zero} = \lambda s.\lambda z.z \textbf{ in}\\
\quad b_1(b_0(b_0(b_1(b_1(b_1(\mathit{zero})))))) \,\mathtt{a}\,\mathtt{c}
\end{array}$$

The part $b_1(b_0(b_0(b_1(b_1(b_1(\mathit{zero}))))))$ corresponds to the binary representation 111001 of 57, with the least significant bit first.

The last line can be replaced with:

$$\begin{array}{l}
\textbf{let } \mathit{twice} = \lambda f.\lambda x.f(f(x)) \textbf{ in}\\
\textbf{let } \mathit{thrice} = \lambda f.\lambda x.f(f(f(x))) \textbf{ in } b_1(\mathit{twice}\,b_0(\mathit{thrice}\,b_1(\mathit{zero})))\,\mathtt{a}\,\mathtt{c}
\end{array}$$

$b_1(\mathit{twice}\,b_0(\mathit{thrice}\,b_1(\mathit{zero})))$ then corresponds to the run-length coding of the binary representation 111001.

□

2.2 Typing

We considered the untyped λ-calculus above, but we can actually assume that any program that generates a tree is well-typed in the intersection type system given below. The assumption that programs are well-typed is important for the program transformations discussed in Section 4. The use of intersection types is important for guaranteeing that we do not lose any expressive power for expressing finite trees: see Theorem 1 below.

The set of *(intersection) types*, ranged over by τ, is given by:

$$\tau ::= \mathtt{o} \mid \tau_1 \wedge \cdots \wedge \tau_k \to \tau$$

Here, k may be 0, in which case we write $\top \to \tau$ for $\tau_1 \wedge \cdots \wedge \tau_k \to \tau$. We assume that $\wedge$ binds tighter than $\to$ and $\to$ is right-associative, so that $\mathtt{o} \wedge \mathtt{o} \to \mathtt{o} \to \mathtt{o}$ means $(\mathtt{o} \wedge \mathtt{o}) \to (\mathtt{o} \to \mathtt{o})$, not $(\mathtt{o} \wedge (\mathtt{o} \to \mathtt{o})) \to \mathtt{o}$. We require that in $\tau_1 \wedge \cdots \wedge \tau_k \to \tau$, $\tau_1, \ldots, \tau_k$ are (syntactically) different from each other. For example, $\mathtt{o} \wedge \mathtt{o} \to \mathtt{o}$ is not allowed. Intuitively, o describes a tree, and $\tau_1 \wedge \cdots \wedge \tau_k \to \tau$ describes a function that takes an element having all of the types $\tau_1, \ldots, \tau_k$, and returns an element of type τ. We sometimes write $\bigwedge_{i \in \{1,\ldots,k\}} \tau_i \to \tau$ for $\tau_1 \wedge \cdots \wedge \tau_k \to \tau$. We also write $\mathsf{o}^k \to \mathsf{o}$ for $\underbrace{\mathsf{o} \to \cdots \to \mathsf{o}}_{k} \to \mathsf{o}$.

A *type environment* is a finite set of type bindings of the form $x:\tau$. Unlike ordinary type environments, we allow multiple occurrences of the same variable, like $\{x : \mathsf{o} \to \mathsf{o}, x : (\mathsf{o} \to \mathsf{o}) \to (\mathsf{o} \to \mathsf{o})\}$. We often omit $\{\,\}$ and just write $x_1 : \tau_1, \ldots, x_n : \tau_n$ for $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$. We write $dom(\Gamma)$ for the set of variables that occur in Γ, i.e., for the set $\{x \mid x : \tau \in \Gamma\}$.

The type judgment relation is of the form $\Gamma \vdash M : \tau$ where Γ is a type environment. It is inductively defined by the following typing rules:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad\qquad \frac{\Sigma(a) = k}{\Gamma \vdash a : \mathsf{o}^k \to \mathsf{o}}$$

$$\frac{\Gamma, x : \tau_1, \ldots, x : \tau_n \vdash M : \tau \qquad x \notin dom(\Gamma)}{\Gamma \vdash \lambda x.M : \tau_1 \wedge \cdots \wedge \tau_n \to \tau}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \wedge \cdots \wedge \tau_n \to \tau \qquad \forall i \in \{1, \ldots, n\}.\Gamma \vdash M_2 : \tau_i}{\Gamma \vdash M_1 M_2 : \tau}$$

Please note that in the rule for applications, n can be 0, in which case M_2 need not be typed.

Example 3 Let $\Sigma = \{\mathsf{c} \mapsto 0\}$, $\tau_0 = \mathsf{o} \to \mathsf{o}$ and $\tau_1 = \tau_0 \to \tau_0$. $((\lambda x.xx)\lambda y.y)\mathsf{c}$ is typed as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{x : \tau_1, x : \tau_0 \vdash x : \tau_1 \quad x : \tau_1, x : \tau_0 \vdash x : \tau_0}{x : \tau_1, x : \tau_0 \vdash xx : \tau_0}}{\emptyset \vdash \lambda x.xx : \tau_1 \wedge \tau_0 \to \tau_0} \quad \dfrac{x : \tau_0 \vdash x : \tau_0}{\emptyset \vdash \lambda x.x : \tau_1} \quad \dfrac{x : \mathsf{o} \vdash x : \mathsf{o}}{\emptyset \vdash \lambda x.x : \tau_0}}{\emptyset \vdash (\lambda x.xx)\lambda y.y : \mathsf{o} \to \mathsf{o}} \qquad \emptyset \vdash \mathsf{c} : \mathsf{o}}{\emptyset \vdash ((\lambda x.xx)\lambda y.y)\mathsf{c} : \mathsf{o}}$$

We can also type a weakly-normalizing but not strongly-normalizing term. Let Ω be $(\lambda x.x\,x)(\lambda x.x\,x)$. Then we have $\emptyset \vdash (\lambda x.\mathsf{c})\Omega : \mathsf{o}$ by:

$$\dfrac{\dfrac{\emptyset \vdash \mathsf{c} : \mathsf{o}}{\emptyset \vdash \lambda x.\mathsf{c} : \top \to \mathsf{o}}}{\emptyset \vdash (\lambda x.\mathsf{c})\Omega : \mathsf{o}}$$

$\square$

Example 4 Recall $M_{2,n} = (\lambda f.\underbrace{f\,f\,\cdots\,f}_{n}\,\mathsf{a}\,\mathsf{c})(\lambda g.\lambda x.g(g(x)))$ in Example 1. Let $\tau_0 = \mathsf{o} \to \mathsf{o}$ and $\tau_{k+1} = \tau_k \to \tau_k$. Then, we have:

$$f : \tau_1, \cdots, f : \tau_n \vdash \underbrace{f\,f\,\cdots\,f}_{n}\,\mathsf{a}\,\mathsf{c} : \mathsf{o}$$
$$\emptyset \vdash \lambda g.\lambda x.g(g(x)) : \tau_k \text{ (for each } k \in \{1, \ldots, n\}).$$

(The first judgment is obtained by assigning type τ_k to the k-th rightmost occurrence of f.) From the first judgment, we have:

$$\emptyset \vdash \underbrace{f\, f\, \cdots\, f}_{n}\, \mathtt{a}\, \mathtt{c} : \tau_1 \wedge \cdots \wedge \tau_n \rightarrow \mathtt{o}.$$

Thus, we have $\emptyset \vdash M_{2,n} : \mathtt{o}$. □

It follows from the standard argument for intersection types [2, 50] that any program that generates a (finite) tree is well-typed, and conversely, any well-typed program of type $\mathtt{o}$ generates a (finite) tree.

Theorem 1 *Let M be a λ-term. Then, $\emptyset \vdash M : \mathtt{o}$ if and only if there exists a tree T such that $T = [\![M]\!]$.*

Proof This follows from standard results on intersection types [2, 49, 50].

For the "only if" part, we first note the facts (i) $\emptyset \vdash M : \mathtt{o}$ implies that M has a β-normal form ([49], Theorem 2.1.14), (ii) if M is in β-normal form and $\emptyset \vdash M : \mathtt{o}$, then M is a tree, and (iii) typing is preserved by β-reductions ([49], Corollary 2.1.11). Fact (ii) follows by easy induction on the size of M: If M is a β-normal form and $\emptyset \vdash M : \mathtt{o}$, then M must be of the form $a\, M_1 \cdots M_k$ with $\Sigma(a) = k$ and $\emptyset \vdash M_i : \mathtt{o}$ for $i \in \{1, \ldots, k\}$. By the induction hypothesis, $M_1, \ldots, M_k$ must be trees, so that M is also a tree. Now, suppose $\emptyset \vdash M : \mathtt{o}$. By (i), there exists a β-normal form $[\![M]\!]$ of M. By (iii), $\emptyset \vdash [\![M]\!] : \mathtt{o}$. By (ii), $[\![M]\!]$ is a tree as required. Appendix A shows a self-contained proof using syntactic techniques.

To show the "if" part, we note the facts: (iv) if M is a tree, then $\emptyset \vdash M : \mathtt{o}$, and (v) typing is preserved by β-expansions (i.e. $\emptyset \vdash N : \tau$ and $M \longrightarrow_\beta N$ imply $\emptyset \vdash M : \tau$) ([49], Corollary 2.1.11). Fact (iv) follows by straightforward induction on the tree M. Now, suppose $[\![M]\!]$ exists and it is a tree. By (iv), $\emptyset \vdash [\![M]\!] : \mathtt{o}$. By (v), we have $\emptyset \vdash M : \mathtt{o}$ as required. □

Example 5 By Theorem 1 above, diverging terms such as $\Omega = (\lambda x.x\, x)(\lambda x.x\, x)$ and $(\lambda x.\mathtt{a}\,(x\, x))(\lambda x.\mathtt{a}\,(x\, x))$ cannot be typed, although the latter generates an *infinite* tree.

As we consider only programs representing (finite) trees, thanks to the theorem above, we can safely assume that all the programs in consideration are well-typed (in the intersection type system above) in the rest of this article.

Remark 1 Instead of the λ-calculus with intersection types, one may use the simply-typed λ-calculus as a data compression language. The simply-typed λ-calculus also enables a hyper-exponential compression ratio. Recall $M_{2,n}$ discussed in Examples 1 and 4. It is not simply-typed, but the same tree can be generated by the following simply-typed λ-term:

$$(\lambda f_1.\cdots \lambda f_n.f_n\, \cdots\, f_1\, \mathtt{a}\, \mathtt{c})\, \underbrace{(\lambda g.\lambda x.g(g(x))) \cdots (\lambda g.\lambda x.g(g(x)))}_{n}.$$

Whether we allow intersection types or not, however, makes the following big differences in theoretical properties:

- The theoretical optimality (with respect to Kolmogorov complexity discussed in Section 2.3) is lost if types are restricted to simple ones.

– The set $\{M \mid [\![M]\!] = T$ and M is typed with intersection types$\}$ is recursively enumerable but not recursive. Thus, the optimality of data representation is undecidable in the presence of intersection types. On the other hand, the set $\{M \mid [\![M]\!] = T$ and M is simply-typed$\}$ is recursive; note that whether a given term is simply-typed is decidable, and any simply-typed term is strongly normalizing. Therefore, the optimality of data representation is decidable if types are restricted to simple ones. In fact, the following simple algorithm always terminates and finds the smallest term M such that $[\![M]\!] = T$.

```
let compressST(T, Candidates) =
  let M :: Candidates' = Candidates in
    if simply-typed(M) and [[M]] = T then M
    else compressST(T, Candidates')
in compressST(T, genterms(T))
```

  Here, genterm(T) returns a list consisting of all the terms no larger than T, sorted in the increasing order of term size.

In practice, an advantage of the restriction to the simply-typed λ-calculus is not so clear. First, even the simply-typed λ-calculus is too powerful for its expressive power to be fully exploited. In the algorithm above, the problem of deciding the equality $[\![M]\!] = T$ is non-elementary [41, 45]. Thus, in practice, other restrictions, such as bounding the order of types (where the order is defined by $\mathit{order}(\mathtt{o}) = 0$, $\mathit{order}(\tau_1 \wedge \cdots \wedge \tau_k \to \tau) = \max(\{\mathit{order}(\tau_i) + 1 \mid i \in \{1, \ldots, k\}\} \cup \{\mathit{order}(\tau)\})$) may be more useful. Secondly, the restriction to the simply-typed λ-calculus seems to forbid some natural compression: recall the simply-typed version of $M_{2,n}$ above, where the term $\lambda g.\lambda x.g(g(x))$ had to be duplicated just because of the difference in types. □

2.3 Relationship with Kolmogorov Complexity

As already discussed in Section 1, the FPCD approach provides a universal compression scheme, in the sense that any compressed data can be expressed in the form of (typed) λ-terms. As sketched below, our representation of compressed data in the form of λ-terms is optimal with respect to Kolmogorov complexity, up to an additive constant [28, 29]. A reader not familiar with Kolmogorov complexity may wish to consult [28, 29].

Let U be a plain universal Turing machine, where the input tape alphabet of the simulated machine is $\{0, 1, \#\}$ and # is used as the delimiter of an input [29]. (Plain) Kolmogorov complexity [28, 29] of a binary string (an element of $\{0, 1\}^*$) v, written $K(v)$, is defined by:

$$K(v) := \min\{|w| \mid w \in \{0, 1\}^*, U(w) = v\}.$$

Here, $|w|$ is the length of w.

Let $B_M \in \{0, 1\}^*$ be a self-delimiting binary code of λ-term M (so that there is no λ-terms M and N such B_M is a proper prefix of B_N). Tromp's coding [47], for example, satisfies this property. Define $K_\lambda(v)$ by:

$$K_\lambda(v) := \min\{|B_M w| \mid [\![M\hat{w}]\!] = \hat{v}\}.$$

Here, $\hat{w}$ is an encoding of a binary string w into a λ-term.

Since the λ-calculus is Turing complete, there exists a λ-term U_λ such that $U(w) = v$ if and only if $[\![U_\lambda(\hat{w})]\!] = \hat{v}$. Thus, $K_\lambda(v) \leq \min\{|B_{U_\lambda} w| \mid U(w) = v\} = |B_{U_\lambda}| + K(v)$. As $|B_{U_\lambda}|$ is independent of v, the result implies that our FPCD approach (combined with a binary coding of λ-terms) achieves an optimal compression size up to an additive constant.

2.4 Relationship with Grammar-based Compression

Grammar-based compression schemes, in which a string or a tree is expressed as a grammar that generates it, have been actively studied recently [4, 18, 30, 37]. Our compression scheme using the λ-calculus can *naturally* mimic grammar-based compression schemes. For example, consider the compression scheme using context-free grammars (with the restriction of cycle-freeness) or straight-line programs. A string s is expressed as a grammar of the following form:

$$X_1 = e_1, X_2 = e_2, \cdots, X_n = e_n,$$

where e_i is either a terminal symbol a, or $X_j X_k$ with $1 \leq j, k < i$, and X_n is the start symbol. It can be expressed as

$$\begin{array}{l} \textbf{let } X_1 = \lambda y.e_1^{(y)} \textbf{ in let } X_2 = \lambda y.e_2^{(y)} \textbf{ in } \cdots \\ \quad \textbf{let } X_n = \lambda y.e_n^{(y)} \textbf{ in } X_n(\mathtt{e}), \end{array}$$

where $e^{(y)}$ is defined by: $a^{(y)} = a(y)$ and $(X_j X_k)^{(y)} = X_j(X_k(y))$. It generates s in the form of a linear tree, with $\mathtt{e}$ as an end-marker. Note that each substring $a_1 \cdots a_k$ is expressed as a function of type $\mathtt{o} \to \mathtt{o}$, which takes (the tree-representation of) a string s' that follows it and, returns $a_1(\cdots a_k(s') \cdots)$. We can also express various extensions of straight-line programs, such as context-free tree grammars [4] and collage systems [18] as λ-terms.

Example 6 Fibonacci words[1] are variations of Fibonacci numbers, obtained by replacing the addition + with the string concatenation, and the first and second elements with $\mathtt{b}$ and $\mathtt{a}$. The n-th word is expressed by the following straight-line program:

$$X_0 = \mathtt{b}, X_1 = \mathtt{a}, X_2 = X_1 X_0, \ldots, X_n = X_{n-1} X_{n-2}.$$

It is encoded as:

$$\begin{array}{l} \textbf{let } X_0 = \mathtt{b} \textbf{ in let } X_1 = \mathtt{a} \textbf{ in let } X_2 = \lambda x.X_1(X_0(x)) \textbf{ in } \cdots \\ \textbf{let } X_n = \lambda x.X_{n-1}(X_{n-2}(x)) \textbf{ in } X_n(\mathtt{e}) \end{array}$$

For $n = 2^m$, we have a more compact encoding:

$$\begin{array}{l} \textbf{let } \mathit{concat} = \lambda x.\lambda y.\lambda z.x(y(z)) \textbf{ in } \textbf{ let } g = \lambda k.\lambda x.\lambda y.k\ y\ (\mathit{concat}\, y\, x) \textbf{ in} \\ \quad \underbrace{\mathit{twice}(\cdots(\mathit{twice}}_{m}(g))\cdots)\ (\lambda x.\lambda y.x)\ \mathtt{b}\ \mathtt{a}\ \mathtt{e} \end{array}$$

A similar encoding is also possible for an arbitrary number n by using b_0 and b_1 in Example 2. □

[1] Fibonacci words and its generalization called *Sturmian words* have been studied in a field called *Stringology* [8].

3 Compression as β-Expansions

In the previous section, we have introduced a typed λ-calculus as a language for representing compressed data, and shown that it allows optimal compression up to an additive constant. As discussed in Remark 1, however, there is no terminating algorithm that takes a tree as an input and outputs its optimal representation. Such an algorithm exists if the language is restricted to simply-typed λ-calculus, but the algorithm is still unrealistic to be used in practice. We describe below an (arguably) more realistic tree compression algorithm, which, given a tree T, finds a small λ-term M (well-typed under the intersection type system) such that $[\![M]\!] = T$. Although it does not guarantee the optimality of the output, it has the following interesting features.

- Simplicity: each step of a compression can be regarded as the inverse of β-reduction, followed by simplification of λ-terms.
- Reuse of existing tree compression algorithms: The algorithm is parametrized by a tree compression algorithm and repeatedly applies it by viewing λ-terms as trees. Therefore, it can be made at least as good as any grammar-based algorithm in terms of the compression ratio (except some overhead caused by the λ-calculus representation), by employing it as the tree compression algorithm and representing its output as a λ-term as discussed in Section 2.4.
- Hyper-exponential compression ratio (in the best case): The algorithm achieves hyper-exponential compression ratio for certain inputs. It also takes advantage of intersection types and outputs terms that are not necessarily simply-typed (see Example 8 below).

The algorithm is still too slow to be used for compression of large data, and it is left for future work to find an algorithm that achieves a better balance between the efficiency and the compression ratio.

We reuse existing algorithms for (context-free) grammar-based tree compression [4, 30], by regarding a λ-term as a term tree (identified up to α-conversion) as follows.

$$x^\sharp = x \quad a^\sharp = a \quad (\lambda x.M)^\sharp = \begin{array}{c}\lambda x\\ |\\ M^\sharp\end{array} \quad (MN)^\sharp = \begin{array}{c}@\\ \diagup\;\;\diagdown\\ M^\sharp \quad N^\sharp\end{array}$$

As we have seen in Section 2.4, compressed data in the form of a context-free grammar can be easily translated to a λ-term. Thus, a grammar-based tree compression algorithm can be regarded as an algorithm for compression of λ-terms. By repeatedly applying such an algorithm to an initial tree T, we can obtain a small λ-term M such that $[\![M]\!] = T$. (There is, however, no guarantee that the resulting term is the smallest such M.) Note that the repeated applications are possible because the input and output languages for the compression algorithm are the same: the λ-calculus.

Figure 3 shows our algorithm, parametrized by two auxiliary algorithms: *compressAsTree* and *simplify*. Given a λ-term M (or a tree as a special case), we just invoke a tree compression algorithm to obtain compressed data in the form of λ-term M_1. It is then simplified by using properties of λ-terms (such as the η-equality). We repeat these steps until the size of a term can no longer be reduced. (In the actual implementation, *compressAsTree* returns multiple candidates, which are inspected for further compression in a breadth-first manner. The termination condition $\#M_2 \geq \#M$ is also relaxed to deal with the case where the term size does not monotonically decrease: See Section 5.)

$compressTerm(M) =$
 let $M_1 = compressAsTree(M)$ **in**
 let $M_2 = simplify(M_1)$ **in**
 if $\#M_2 \geq \#M$ **then** M **else** $compressTerm(M_2)$

Fig. 3 Compression algorithm for λ-terms.

Because of the repeated applications of *compressAsTree*, we can actually use the following very simple algorithm for *compressAsTree*, which just finds and extracts a common tree context, rather than more sophisticated algorithms [4, 30]. Let us define a context with (up to) k-holes by:

$$C ::= [\,]_1 \mid \cdots \mid [\,]_k \mid x \mid a \mid CC \mid \lambda x.C$$

We write $C[M_1, \ldots, M_k]$ for the term obtained by replacing each $[\,]_i$ in C with M_i. Note that ignoring binders, a context is just a tree context with up to k holes. Then, *compressAsTree* just needs to find (non-deterministically) contexts C_0, C_1, C_2, C_3 and terms $M_1, \ldots, M_k, N_1, \ldots, N_k$ such that

- (i) $M = C_0[C_1[C_2[M_1, \ldots, M_k], C_2[N_1, \ldots, N_k]]]$ or (ii) $M = C_0[C_1[C_2[M_1, \ldots, M_k]]] \wedge M_i = C_3[C_2[N_1, \ldots, N_k]]$;
- the free variables in $M_1, \ldots, M_k, N_1, \ldots, N_k$ are not bound in C_2, and the free variables in C_2 are not bound in C_1; and
- every hole of C_0, C_1, C_2 occurs at least once. (For example, if C_2 is a context with two holes, C_2 must contain both $[\,]_1$ and $[\,]_2$ at least once.)

Here, in the first condition above, (i) and (ii) are the cases where the common context C_2 occurs horizontally and vertically, respectively: see Figure 4. The output is:

$$C_0[(\lambda f.C_1[f\,M_1\,\cdots\,M_k, f\,N_1\,\cdots\,N_k])(\lambda\tilde{x}.C_2[\tilde{x}])]$$

in case (i), and

$$C_0[(\lambda f.C_1[f\,M_1\,\cdots\,M_{i-1}\,M_i'\,M_{i+1}\,\cdots\,M_k])(\lambda\tilde{x}.C_2[\tilde{x}])]$$

where $M_i' = C_3[f\,N_1\,\cdots\,N_k]$ in case (ii), and $\tilde{x}$ denotes the sequence $x_1, \ldots, x_k$.

The third condition above ensures that no vacuous λ-abstractions are introduced. For example, if we allowed a two-hole context $C_2 = [\,]_2$ (which does not contain $[\,]_1$), $\lambda x_1.\lambda x_2.x_2$ would be introduced by the transformation above.

The transformation above is a restricted form of β-expansion step: $C[[N/x]M] \longrightarrow C[(\lambda x.M)N]$, applicable only when M contains two occurrences of x.

The sub-procedure *compressAsTree* above is highly non-deterministic in the choice of contexts. In our prototype implementation, we pick every pair (M', M'') of subterms of M and find the maximum common context C_2 such that $M' = C_2[M_1, \ldots, M_k]$ and $M'' = C_2[N_1, \ldots, N_k]$. For splitting the enclosing context into C_0 and C_1, we choose the largest[2] C_1 that satisfies the condition on bound variables. The resulting procedure is still non-deterministic in the choice of the pairs (M', M''), and our implementation applies the depth-first search. See Section 5 for more details.

For the simplification procedure *simplify*, we apply the following rules until no more rules become applicable.

[2] It may also make sense to choose the *smallest* one instead.

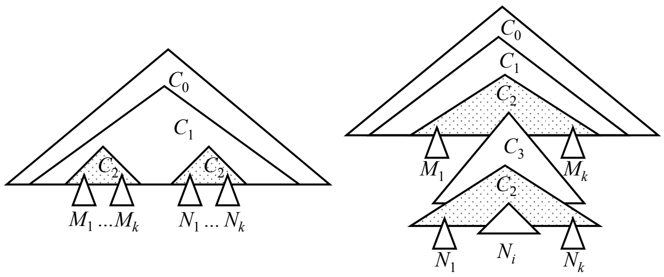

Fig. 4 Cases where the common context C_2 occurs horizontally (left, case (i)), and vertically (right, case (ii)).

- η-conversion: $\lambda x.M\,x \longrightarrow M$ if x is not free in M.
- β-reduction when the argument is a variable: $(\lambda x.M)y \longrightarrow [y/x]M$.
- β-reduction for linear functions: $(\lambda x.M)N \longrightarrow [N/x]M$ if x occurs (syntactically) at most once in M.

Remark 2 The output of our compression algorithm above belongs to λ-I calculus [6], where each λ-abstraction $\lambda x.M$ contains at least one occurrence of x in M. To observe it, recall that *compressAsTree* applies the β-expansion $C[[N/x]M] \longrightarrow C[(\lambda x.M)N]$ only when M contains two occurrences of x. The three simplification transformations given above also preserve the property that each bound variable occurs at least once.

By the above observation, we know that the term in Example 2 cannot be generated by our compression algorithm, since the term contains a vacuous λ-abstraction **let** *zero* $= \lambda s.\lambda z.z$ **in** (where s does not occur in the body of λs). The following slight variation (obtained by replacing b_1 *zero* by $\lambda s.s$) can however be obtained by our algorithm:

$$\begin{array}{l}\textbf{let } b_0 = \lambda n.\lambda s.\lambda z.n\,s\,(n\,s\,z) \textbf{ in}\\ \textbf{let } b_1 = \lambda n.\lambda s.\lambda z.s\,(n\,s\,(n\,s\,z)) \textbf{ in}\\ \quad b_1(b_0(b_0(b_1(b_1(\lambda s.s)))))\,\mathtt{a}\,\mathtt{c}\end{array}$$

□

Example 7 Consider a tree $\mathtt{a}^9(\mathtt{c})$. Let $C_0, C_1, C_2, C_3, M_1, N_1$ be:

$$C_0 = [\,]_1 \quad C_1 = [\,]_1 \quad C_2 = \mathtt{a}^3[\,]_1 \quad C_3 = [\,]_1 \quad M_1 = \mathtt{a}^6(c) \qquad N_1 = \mathtt{a}^3(c)$$

Then, case (ii) applies and the following term is obtained:

$$(\lambda f.f(f(\mathtt{a}^3(c))))\lambda x.\mathtt{a}^3(x)$$

Next, we again extract the common context $\mathtt{a}^3[\,]_1$, and obtain

$$(\lambda g.(\lambda f.f(f(g(c))))\lambda x.g\,x)(\lambda x.\mathtt{a}^3(x))$$

By using the η-equality $\lambda x.g\,x = g$, we get:

$$(\lambda g.(\lambda f.f(f(g(c))))g)(\lambda x.\mathtt{a}^3(x)).$$

As a part of the simplification procedure, we also β-reduce terms of the form $(\lambda x.M)y$ and obtain: $(\lambda g.g(g(g(c))))(\lambda x.\mathtt{a}^3(x))$. In the third iteration, we can extract the common context $[\,]_1([\,]_1([\,]_1\,[\,]_2))$ and obtain $(\lambda h.(\lambda g.h\,g\,\mathtt{c})(\lambda x.h\,\mathtt{a}\,x))(\lambda f.\lambda x.f(f(f\,x)))$. By simplifying the term (by η-conversion and β-reduction for the linear function $\lambda g.h\,g\,\mathtt{c}$), we obtain:

$$(\lambda h.(h\,(h\,\mathtt{a})\,\mathtt{c}))(\lambda f.\lambda x.f(f(f\,x))). \qquad \square$$

Example 8 We give an example for which our compression algorithm generates a term that is not simply-typed (but well-typed in the intersection type system). Consider the tree $\mathtt{a}^4(\mathtt{c})$. By extracting the common context $\mathtt{a}^2[\,]_1$, we obtain:

$$(\lambda f.(f(f\,\mathtt{c})))(\lambda x.\mathtt{a}(\mathtt{a}\,x)).$$

By extracting the context $[\,]_1([\,]_1\,[\,]_2)$, we further obtain:

$$(\lambda g.(\lambda f.g\,f\,\mathtt{c})(\lambda x.g\,\mathtt{a}\,x))(\lambda h.\lambda x.h(h\,x)),$$

which can be simplified as follows.

$$\begin{array}{l}(\lambda g.(\lambda f.g\,f\,\mathtt{c})(\underline{\lambda x.g\,\mathtt{a}\,x}))(\lambda h.\lambda x.h(h\,x))\\ \longrightarrow_\eta (\lambda g.\underline{(\lambda f.g\,f\,\mathtt{c})(g\,\mathtt{a})})(\lambda h.\lambda x.h(h\,x))\\ \longrightarrow_\beta (\lambda g.g\,(g\,\mathtt{a})\,\mathtt{c})(\lambda h.\lambda x.h(h\,x)).\end{array}$$

By extracting the context $[\,]_1([\,]_1\,[\,]_2)$ again, we obtain:

$$(\lambda f.(\lambda g.f\,g\,\mathtt{a}\,\mathtt{c})(\lambda h.\lambda x.f\,h\,x))(\lambda k.\lambda x.k(k\,x)),$$

which is simplified as follows.

$$\begin{array}{l}(\lambda f.(\lambda g.f\,g\,\mathtt{a}\,\mathtt{c})(\underline{\lambda h.\lambda x.f\,h\,x}))(\lambda k.\lambda x.k(k\,x))\\ \longrightarrow_\eta^* (\lambda f.\underline{(\lambda g.f\,g\,\mathtt{a}\,\mathtt{c})f})(\lambda k.\lambda x.k(k\,x))\\ \longrightarrow_\beta (\lambda f.f\,f\,\mathtt{a}\,\mathtt{c})(\lambda k.\lambda x.k(k\,x)).\end{array}$$

This is $M_{2,2}$ in Example 1.

We show informally that $M_{2,n}$ in Example 1 is obtained from $\mathtt{a}^{2\uparrow\uparrow n}\mathtt{c}$, where $2\uparrow\uparrow n = \overbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}^{n}$, by induction on n. By repeatedly extracting the context $[\,]_1([\,]_1\,[\,]_2)$ (in a manner to similar to the transformation of $\mathtt{a}^4\,\mathtt{c}$ to $(\lambda g.g\,(g\,\mathtt{a})\,\mathtt{c})(\lambda h.\lambda x.h(h\,x))$ above), we obtain

$$(\lambda g.\,\underbrace{g(g(\cdots g}_{2\uparrow\uparrow(n-1)}(\mathtt{a})\cdots)\mathtt{c})\lambda h.\lambda x.h(h\,x).$$

By induction hypothesis, the part $\underbrace{g(g(\cdots g}_{2\uparrow\uparrow(n-1)}(\mathtt{a})\cdots)(= g^{2\uparrow\uparrow(n-1)}\,\mathtt{a})$ can be compressed to $(\lambda f.\underbrace{f\,f\,\cdots\,f}_{n-1}\,g\,\mathtt{a})\lambda h.\lambda x.h(h\,x)$. Thus, we obtain:

$$(\lambda g.(\lambda f.\underbrace{f\,f\,\cdots\,f}_{n-1}\,g\,\mathtt{a})(\lambda h.\lambda x.h(h\,x))\mathtt{c})\lambda h.\lambda x.h(h\,x).$$

By extracting the common term $\lambda h.\lambda x.\lambda h(h\,x)$, we obtain

$$(\lambda k.(\lambda g.(\lambda f.\underbrace{f\,f\,\cdots\,f}_{n-1}\,g\,\mathtt{a})k\,\mathtt{c})k)\lambda h.\lambda x.h(h\,x),$$

which can be simplified as follows.

$$\begin{aligned}
&(\lambda k.(\lambda g.\underline{(\lambda f.\overbrace{f\,f\,\cdots\,f}^{n-1}\,g\,\mathtt{a})k}\,\mathtt{c})k)\lambda h.\lambda x.h(h\,x)\\
\longrightarrow_\beta\ &(\lambda k.\underline{(\lambda g.\overbrace{k\,k\,\cdots\,k}^{n-1}\,g\,\mathtt{a}\,\mathtt{c})k})\lambda h.\lambda x.h(h\,x)\\
\longrightarrow_\beta\ &(\lambda k.\underbrace{k\,k\,\cdots\,k}_{n}\,\mathtt{a}\,\mathtt{c})\lambda h.\lambda x.h(h\,x).
\end{aligned}$$

Thus, we have obtained $M_{2,n}$. □

Example 9 Recall the tree in Figure 1. By extracting the first two occurrences of the common context (with zero holes) $\mathtt{c\,a\,a}$, we obtain:

$$(\lambda x.\mathtt{c}\,x\,(\mathtt{d}\,x\,(\mathtt{c}\,(\mathtt{c\,a\,a})\,(\mathtt{d}\,(\mathtt{c\,a\,a})\,(\mathtt{c\,a\,a})))))(\mathtt{c\,a\,a}).$$

By further extracting the common context $\mathtt{c\,a\,a}$ repeatedly (and applying *simplify*), we get $(\lambda C.\mathtt{c}\,C\,(\mathtt{d}\,C\,(\mathtt{c}\,C\,(\mathtt{d}\,C\,C))))(\mathtt{c\,a\,a})$. By extracting the common context $\mathtt{a}$, we obtain

$$(\lambda C.\mathtt{c}\,C\,(\mathtt{d}\,C\,(\mathtt{c}\,C\,(\mathtt{d}\,C\,C))))((\lambda A.(\mathtt{c}\,A\,A))\mathtt{a}).$$

This corresponds to the DAG representation in Figure 1 of [31] and also to the regular grammar representation in Figure 2 of [4]. By extracting the common context $\lambda y.\mathtt{c}\,C\,(\mathtt{d}\,C\,y)$, the term is further transformed to:

$$(\lambda C.(\lambda B.B(B(C)))(\lambda y.\mathtt{c}\,C\,(\mathtt{d}\,C\,y)))((\lambda A.(\mathtt{c}\,A\,A))\mathtt{a}).$$

This corresponds to the sharing graph representation in Figure 1 of [31] and to the CFG representation in Figure 1 of [4]. □

Relationship with CFG-based Tree Compression Algorithms. As demonstrated in Example 9, context-free grammar-based tree compression algorithms [4, 30] can be mimicked by our compression method based on λ-calculus. In fact, they may be viewed as a controlled and restricted form of our compression algorithm. For example, for efficient compression, Busatto et al. [4] impose restrictions on the number of holes and the size of common contexts, and also introduce certain priorities among subterms from which common contexts are searched. (There is also another difference that Busatto's algorithm finds more than two occurrences of a common context at once, but it can be mimicked by repeated applications of *compressAsTree* above.)

A more fundamental restriction of the previous approaches is that they [4] extract only common tree contexts with first-order types, of the form $\mathtt{o} \to \cdots \to \mathtt{o} \to \mathtt{o}$. Because of this difference, our compression algorithm based on the λ-calculus is more powerful than ordinary grammar-based compression algorithms. For example, the compression discussed in Example 7 is not possible with CFG-based compression: note that the context $[\,]_1([\,]_1([\,]_1\,[\,]_2))$ (expressed by $\lambda f.\lambda x.f(f(f\,x))$) extracted during the compression has an order-2 type $(\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}$; therefore, it cannot be shared in the context-free tree grammar approach [4].

Limitations. The compression algorithm sketched above is neither efficient nor complete. By the incompleteness, we mean that there is a λ-term M and a tree T such that $[\![M]\!] = T$ but M cannot be obtained from T by the algorithm.

For example, consider the following tree (represented as a term) T_1:

$$\mathtt{br}\ (\mathtt{a}\,(\mathtt{b}\,(x\,(y\,(\mathtt{c}\,(\mathtt{d}\,\mathtt{e})))))) \ (\mathtt{a}\,(\mathtt{b}\,(z\,(\mathtt{c}\,(\mathtt{d}\,\mathtt{e})))))$$

It can be expressed by the following term M:

$$\mathbf{let}\ f = \lambda g.\mathtt{a}\,(\mathtt{b}\,(g\,(\mathtt{c}\,(\mathtt{d}\,\mathtt{e}))))\ \mathbf{in}\ \mathtt{br}\ (f\,\lambda u.x(y\,u))\ (f\,z),$$

but M cannot be obtained by our algorithm. To enable the above compression, we need to β-expand T_1 to:

$$\mathtt{br}\ (\mathtt{a}\,(\mathtt{b}\,((\lambda u.x(y\,u))\,(\mathtt{c}\,(\mathtt{d}\,\mathtt{e}))))) \ (\mathtt{a}\,(\mathtt{b}\,(z\,(\mathtt{c}\,(\mathtt{d}\,\mathtt{e})))))$$

before applying our algorithm. Such pre-processing is however non-trivial in general.

For another example, consider the following tree (represented as a term) T_2:

$$\mathtt{br}\ (\mathtt{a}_1\,(\mathtt{a}_2\ \cdots\ (\mathtt{a}_n\,\mathtt{e})\cdots))\ (\mathtt{a}_n\ \cdots\ (\mathtt{a}_2\,(\mathtt{a}_1\,(\mathtt{e})))\cdots),$$

which consists of a linear tree representing the sequence $\mathtt{a}_1, \dots, \mathtt{a}_n$ and its reverse.

The following term M generates T, but the common pattern h cannot be found by our algorithm.

$$\begin{array}{l}\mathbf{let}\ h = \lambda a.\lambda k.\lambda x.\lambda y.k\,(a\,x)\,(\lambda z.y(a(z)))\ \mathbf{in}\\ \mathbf{let}\ id = \lambda z.z\ \mathbf{in}\\ h\,\mathtt{a}_n(\cdots(h\,\mathtt{a}_2\,(h\,\mathtt{a}_1\,(\lambda x.\lambda y.\mathtt{br}\,x\,(y\,\mathtt{e}))))\cdots)\,\mathtt{e}\,id\end{array}$$

Actually, finding terms h, k, x, y such that

$$[\![h\,\mathtt{a}_n(\cdots(h\,\mathtt{a}_2(h\,\mathtt{a}_1\,k)))\,x\,y]\!] = T_2$$

is an instance of the higher-order matching problem [42]. Thus, higher-order matching algorithms may be applicable to our data compression scheme. We leave for future work a good characterization of the terms obtained by our compression algorithm (besides the characterization by λ-I calculus in Remark 2), as well as extensions (e.g. with higher-order matching) to obtain more powerful and/or efficient algorithms.

4 Processing of Compressed Data

This section discusses how to process compressed data *without decompression*.

4.1 Pattern Matching as Higher-Order Model Checking

We first discuss the problem of answering whether $[\![M]\!]$ matches P, given a program M and a regular tree pattern P. For instance, we may wish to check whether some path from the root of the tree $[\![M]\!]$ contains `ab` as a subpath, or check whether $[\![M]\!]$ contains a subtree of the shape:

$$\begin{array}{ccc} & \mathtt{c} & \\ \mathtt{d} & & \mathtt{d} \\ | & & | \\ \cdots & & \cdots \end{array}$$

Such a pattern matching problem can be formalized as an acceptance problem for tree automata [7].

Below we write $dom(f)$ for the domain of a map f.

Definition 1 (tree automata) *A (top-down, alternating) tree automaton $\mathcal{A}$ is a quadruple (Σ, Q, q_I, Δ), where Σ is a ranked alphabet, Q is a set of states, q_I is the initial state, and $\Delta(\subseteq Q \times dom(\Sigma) \times 2^{\{1,\ldots,m\}\times Q})$ is a transition function (where m is the largest arity of symbols in Σ), such that $(q, a, S \cup \{(i, q')\}) \in \Delta$ implies $1 \le i \le \Sigma(a)$. The reduction relation $V_1 \longrightarrow V_2$ on subsets of $Q \times \mathcal{T}_\Sigma$ is defined by: $V \cup \{(q, a\, T_1\, \cdots\, T_n)\} \longrightarrow V \cup \{(q', T_i) \mid (i, q') \in S\}$ if $(q, a, S) \in \Delta$. A tree T is accepted by $\mathcal{A}$ if $\{(q_I, T)\} \longrightarrow^* \emptyset$. We write $\mathcal{L}(\mathcal{A})$ for the set of trees accepted by $\mathcal{A}$.*

Example 10 Consider the automaton $\mathcal{A}_1 = (\Sigma_1, \{q_0, q_1\}, q_0, \Delta)$ where $\Sigma_1 = \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{e} \mapsto 0\}$ and Δ is given by:

$$\Delta = \{(q_0, \mathtt{a}, \{(1, q_1)\}), (q_0, \mathtt{b}, \{(1, q_0)\}), (q_1, \mathtt{a}, \{(1, q_1)\}), (q_1, \mathtt{b}, \emptyset)\}$$

Then, a Σ_1-labeled tree T contains a subtree of the form $\mathtt{a}(\mathtt{b}(\cdots))$ if and only if T is accepted by $\mathcal{A}_1$. □

Example 11 Let $\Sigma_2 = \{\mathtt{b} \mapsto 1, \mathtt{c} \mapsto 2, \mathtt{d} \mapsto 1, \mathtt{e} \mapsto 0\}$. Consider the automaton $\mathcal{A}_2 = (\Sigma_2, \{q_0, q_1\}, q_0, \Delta_2)$ where Δ_2 is given by:

$$\begin{aligned}\Delta_2 = \{&(q_0, \mathtt{b}, \{(1, q_0)\}), (q_0, \mathtt{c}, \{(1, q_1), (2, q_1)\}), (q_0, \mathtt{c}, \{(1, q_0)\}),\\ &(q_0, \mathtt{c}, \{(2, q_0)\}), (q_0, \mathtt{d}, \{(1, q_0)\}), (q_1, \mathtt{d}, \emptyset)\}\end{aligned}$$

Then, a Σ_2-labeled tree T contains a subtree of the form $\mathtt{c}\,(\mathtt{d}\,\cdots)\,(\mathtt{d}\,\cdots)$ if and only if T is accepted by $\mathcal{A}_2$. □

Remark 3 The definition of alternating tree automata above is different from the standard definition [7], where the transition function is defined as a map from $Q \times dom(\Sigma)$ to positive boolean formulas constructed from atomic formulas of the form (i, q). For instance, Δ_2 in Example 11 is defined as:

$$\begin{array}{ll} \Delta_2(q_0, \mathtt{b}) = (1, q_0) & \Delta_2(q_1, \mathtt{b}) = \mathbf{false} \\ \Delta_2(q_0, \mathtt{c}) = ((1, q_1) \wedge (2, q_1)) \vee (1, q_0) \vee (2, q_0) & \Delta_2(q_1, \mathtt{c}) = \mathbf{false} \\ \Delta_2(q_0, \mathtt{d}) = (1, q_0) & \Delta_2(q_1, \mathtt{d}) = \mathbf{true} \end{array}$$

Both the definitions are equivalent in the expressive power although the *size* of the descriptions of automata can be different; our definition corresponds to the case where the image of transition functions is restricted to formulas in disjunctive normal form. Our definition is motivated to make the definitions of tree automata and tree transducers (introduced in Definition 3) similar. □

The goal here is, given a (well-typed) program M and an automaton $\mathcal{A}$, to check whether $[\![M]\!] \in \mathcal{L}(\mathcal{A})$ holds. A simple decision algorithm is to decompress M (i.e. fully β-reduce M) to a tree $T\,(= [\![M]\!])$ and run the automaton $\mathcal{A}$ for T. This is however inefficient if T is large or the reduction sequence of M is long. Instead, we use the type-based technique for model checking higher-order recursion schemes [22, 48], to reduce $[\![M]\!] \stackrel{?}{\in} \mathcal{L}(\mathcal{A})$ to a type-checking problem for M. Because of subtle differences between higher-order recursion schemes [19, 34] and the language considered here (see Remark 6), we give a direct construction of the type system below.

Definition 2 (refinement intersection types) *Let $\mathcal{A} = (\Sigma, Q, q_I, \Delta)$ be a tree automaton. The set $\mathbf{RTy}_Q$ of refinement intersection types, ranged over by θ, is given by:*

$$\theta ::= q\,(\in Q) \mid \theta_1 \wedge \cdots \wedge \theta_k \to \theta$$

Here, k may be 0, in which case we write $\top \to \theta$ for $\theta_1 \wedge \cdots \wedge \theta_k \to \theta$. We assume some strict total order $<$ (e.g., the lexicographic order) on refinement intersection types, and require that $\theta_1 < \theta_2 < \cdots < \theta_k$ holds in $\theta_1 \wedge \cdots \wedge \theta_k \to \theta$.

Intuitively, q describes the set of trees accepted by $\mathcal{A}$ from the state q (i.e., accepted by (Σ, Q, q, Δ)). The type $\theta_1 \wedge \cdots \wedge \theta_k \to \theta$ describes a function that takes an element of types $\theta_1, \ldots, \theta_k$, and returns an element of type θ. For example, recall the automaton $\mathcal{A}_2$ in Example 11. The symbol $\mathtt{b}$ has type $q_0 \to q_0$, since $\mathtt{b}(T)$ is accepted from q_0 if T is accepted from q_0. The term $\lambda x.\mathtt{c}\,x\,x$ has type $q_0 \to q_0$, since $\mathtt{c}\,T\,T$ is accepted from q_0 if T is accepted from q_0. It has also types $q_1 \to q_0$.

Remark 4 The restriction on the syntax of refinement intersection types above enforces that the intersection type constructor is essentially idempotent, commutative and associative: there is a unique representation for types that are mutually equivalent with respect to the laws of idempotency, commutativity, and associativity on $\wedge$. Based on the assumption, we sometimes write $\bigwedge_{i \in I} \theta_i \to \theta$ (where I is a finite set of indices) for $\theta'_1 \wedge \cdots \wedge \theta'_k \to \theta$ when $\{\theta_i \mid i \in I\} = \{\theta'_1, \ldots, \theta'_k\}$. In some previous type systems for higher-order model checking [20, 25], we represented an intersection type as a set instead of a sequence, i.e., used the notation $\bigwedge\{\theta_1, \ldots, \theta_k\} \to \theta$ instead of $\theta_1 \wedge \cdots \wedge \theta_k \to \theta$. This automatically enforces that intersection types are idempotent, commutative, and associative. Having the order between $\theta_1, \ldots, \theta_k$ is however important for the development in Section 4.2.1. $\square$

We shall construct below a type system for reasoning about the types of terms. A *refinement type environment* Ψ is a finite set of type bindings of the form $x : \theta$, where multiple occurrences of the same variable are allowed as in the intersection type system in Section 2.2. We write $dom(\Psi)$ for the set $\{x \mid x : \theta \in \Psi\}$ of variables. The type judgment relation $\Psi \vdash_{\mathcal{A}} M : \theta$ is defined by:

$$\frac{}{\Psi, x : \theta \vdash_{\mathcal{A}} x : \theta} \qquad \frac{(q, a, \{(i, q_j) \mid 1 \le i \le \Sigma(a), j \in I_i\}) \in \Delta}{\Psi \vdash_{\mathcal{A}} a : \bigwedge_{j \in I_1} q_j \to \cdots \to \bigwedge_{j \in I_{\Sigma(a)}} q_j \to q}$$

$$\frac{\Psi, x : \theta_1, \ldots, x : \theta_n \vdash_{\mathcal{A}} M : \theta \qquad x \text{ does not occur in } \Psi}{\Psi \vdash_{\mathcal{A}} \lambda x.M : \theta_1 \wedge \cdots \wedge \theta_n \to \theta}$$

$$\frac{\Psi \vdash_{\mathcal{A}} M_1 : \theta_1 \wedge \cdots \wedge \theta_n \to \theta \qquad \forall i \in \{1, \ldots, n\}.\Psi \vdash_{\mathcal{A}} M_2 : \theta_i}{\Psi \vdash_{\mathcal{A}} M_1 M_2 : \theta}$$

Note that these typing rules are the same as those for the intersection type system in Section 2.2 except the rule for constants. The type of a constant a depends on the transition function Δ. The condition $(q, a, \{(i, q_j) \mid 1 \leq i \leq \Sigma(a), j \in I_i\}) \in \Delta$ means that in order for a tree of the form $(a\, T_1 \, \cdots \, T_k)$ to be accepted from state q, it suffices that each T_i is accepted from q_j for all $j \in I_i$, i.e., T_i has type $\bigwedge_{j \in I_i} q_j$. Thus, a can be viewed as a function of type $\bigwedge_{j\in I_1} q_j \to \cdots \to \bigwedge_{j \in I_{\Sigma(a)}} q_j \to q$.

Let us define mappings from refinement types (resp., refinement type environments) to types (resp., type environments) by:

$$\begin{array}{l} \alpha(q) = \mathtt{o} \\ \alpha((\theta_1 \wedge \cdots \wedge \theta_k \to \theta)) = \alpha(\theta_1) \wedge \cdots \wedge \alpha(\theta_k) \to \alpha(\theta) \\ \alpha(\{x_1 : \theta_1, \ldots, x_k : \theta_k\}) = \{x_1 : \alpha(\theta_1), \ldots, x_k : \alpha(\theta_k)\} \end{array}$$

Here, in $\alpha(\theta_1)\wedge\cdots\wedge\alpha(\theta_k)$, we assume that duplicated elements are deleted; for example, $(\mathtt{o} \to \mathtt{o}) \wedge (\mathtt{o} \to \mathtt{o} \to \mathtt{o}) \wedge (\mathtt{o} \to \mathtt{o}) = (\mathtt{o} \to \mathtt{o}) \wedge (\mathtt{o} \to \mathtt{o} \to \mathtt{o})$. The refinement type system above is indeed a refinement of the type system introduced in Section 2 in the following sense.

Lemma 1 *If $\Psi \vdash_{\mathcal{A}} M : \theta$, then $\alpha(\Psi) \vdash M : \alpha(\theta)$.*

Proof This follows by straightforward induction on the derivation of $\Psi \vdash_{\mathcal{A}} M : \theta$. □

The refinement type system is sound and complete for the problem in consideration.

Theorem 2 *Let M be a program and $\mathcal{A} = (\Sigma, Q, q_I, \Delta)$ be a tree automaton. Then, $[\![M]\!] \in \mathcal{L}(\mathcal{A})$ if and only if $\emptyset \vdash_{\mathcal{A}} M : q_I$.*

Proof We use the following facts:

(i) For every Σ-labeled tree T, $T \in \mathcal{L}(\mathcal{A})$ if and only if $\emptyset \vdash_{\mathcal{A}} T : q_I$.
(ii) Typing is preserved by β-reduction, i.e., $\Psi \vdash M : \theta$ and $M \longrightarrow_\beta N$ imply $\Psi \vdash N : \theta$.
(iii) Typing is preserved by β-expansion, i.e., $\Psi \vdash N : \theta$ and $M \longrightarrow_\beta N$ imply $\Psi \vdash M : \theta$.

Fact (i) follows by straightforward inductions on the structure of T. The proofs of (ii) and (iii) are standard [22, 49], hence omitted.

To show the "if" part, suppose $\emptyset \vdash_{\mathcal{A}} M : q_I$. By Lemma 1, we have $\emptyset \vdash M : \mathtt{o}$. By Theorem 1, $[\![M]\!]$ exists. By (ii), $\emptyset \vdash_{\mathcal{A}} [\![M]\!] : q_I$. By (i), we have $[\![M]\!] \in \mathcal{L}(\mathcal{A})$.

To show the "only if" part, suppose $[\![M]\!] \in \mathcal{L}(\mathcal{A})$. By (i), we have $\emptyset \vdash_{\mathcal{A}} [\![M]\!] : q_I$. By (iii), we have $\emptyset \vdash_{\mathcal{A}} M : q_I$ as required. □

Suppose that a derivation for $\emptyset \vdash M : \mathtt{o}$ is given. The result of Tsukada and Kobayashi ([48], Theorem 5) implies that to check whether $\emptyset \vdash_{\mathcal{A}} M : q_I$ holds, we just need to generate a *finite* set of candidates of derivation trees for $\emptyset \vdash_{\mathcal{A}} M : q_I$, and check whether one of them is valid. To state it more formally, we need to introduce some terminologies. The refinement relation $\theta :: \tau$ is defined by:

$$\frac{q \in Q}{q :: \mathtt{o}} \qquad\qquad \frac{\theta :: \tau \qquad \forall j \in \{1, \ldots, m\}. \exists i \in \{1, \ldots, k\}. \theta_j :: \tau_i}{(\theta_1 \wedge \cdots \wedge \theta_m \to \theta) :: (\tau_1 \wedge \cdots \wedge \tau_k \to \tau)}$$

Intuitively, $\theta :: \tau$ holds if θ matches the shape determined by τ. For example, $(q_1 \wedge q_2 \to q) :: (\mathtt{o} \to \mathtt{o})$ holds but $(q_1 \to q_2 \to q) :: (\mathtt{o} \to \mathtt{o})$ does not. The "shape" itself can contain intersection types: $((q_1 \wedge q_2 \to q) \wedge (q_1 \to q) \wedge (q_1 \to q_2 \to q) \to q) :: ((\mathtt{o} \to \mathtt{o}) \wedge (\mathtt{o} \to \mathtt{o} \to \mathtt{o}) \to \mathtt{o})$ holds.

We extend the refinement relation to the relation on type environments by:

$$\Psi :: \Gamma \Leftrightarrow \forall x : \theta \in \Psi . \exists \tau . (x : \tau \in \Gamma \wedge \theta :: \tau).$$

Let π and π' be derivation trees for $\Psi \vdash_{\mathcal{A}} M : \theta$ and $\Gamma \vdash M : \tau$ respectively. π is *a refinement* of π', written $\pi :: \pi'$, if for each node labeled by $\Psi_1 \vdash_{\mathcal{A}} M_1 : \theta_1$ in π, there exists a corresponding node labeled by $\Gamma_1 \vdash M_1 : \tau_1$ in π' such that $\Psi_1 :: \Gamma_1$ and $\theta_1 :: \tau_1$. More precisely, the refinement of derivation trees is inductively defined as follows.

- $\dfrac{}{\Psi, x : \theta \vdash_{\mathcal{A}} x : \theta}$ is a refinement of $\dfrac{}{\Gamma, x : \tau \vdash x : \tau}$ if $\Psi :: \Gamma$ and $\theta :: \tau$.
- $\dfrac{}{\Psi \vdash_{\mathcal{A}} a : \bigwedge_{j \in I_1} q_j \to \cdots \to \bigwedge_{j \in I_k} q_j \to q}$ is a refinement of $\dfrac{}{\Gamma \vdash a : \mathsf{o}^k \to q}$ if $\Psi :: \Gamma$.
- $\dfrac{\pi}{\Psi \vdash_{\mathcal{A}} \lambda x.M : \theta_1 \wedge \cdots \wedge \theta_n \to \theta}$ is a refinement of $\dfrac{\pi'}{\Gamma \vdash \lambda x.M : \tau_1 \wedge \cdots \wedge \tau_k \to \tau}$ if π is a refinement of π' with $\Psi :: \Gamma$ and $(\theta_1 \wedge \cdots \wedge \theta_n \to \theta) :: (\tau_1 \wedge \cdots \wedge \tau_k \to \tau)$.
- $$\dfrac{\dfrac{\pi_0}{\Psi \vdash_{\mathcal{A}} M_1 : \theta_1 \wedge \cdots \wedge \theta_n \to \theta} \quad \dfrac{\pi_1}{\Psi \vdash_{\mathcal{A}} M_2 : \theta_1} \quad \cdots \quad \dfrac{\pi_n}{\Psi \vdash_{\mathcal{A}} M_2 : \theta_n}}{\Psi \vdash_{\mathcal{A}} M_1 M_2 : \theta}$$
  is a refinement of:
  $$\dfrac{\dfrac{\pi'_0}{\Gamma \vdash_{\mathcal{A}} M_1 : \tau_1 \wedge \cdots \wedge \tau_k \to \tau} \quad \dfrac{\pi'_1}{\Gamma \vdash M_2 : \tau_1} \quad \cdots \quad \dfrac{\pi'_k}{\Gamma \vdash M_2 : \tau_k}}{\Psi \vdash M_1 M_2 : \tau}$$
  if $\dfrac{\pi_0}{\Psi \vdash_{\mathcal{A}} M_1 : \theta_1 \wedge \cdots \wedge \theta_n \to \theta}$ is a refinement of $\dfrac{\pi'_0}{\Gamma \vdash M_1 : \tau_1 \wedge \cdots \wedge \tau_k \to \tau}$, and for each $i \in \{1, \ldots, n\}$, there exists $j \in \{1, \ldots, k\}$ such that $\dfrac{\pi_i}{\Psi \vdash_{\mathcal{A}} M_2 : \theta_i}$ is a refinement of $\dfrac{\pi'_j}{\Gamma \vdash M_2 : \tau_j}$.

The following is the result of Tsukada and Kobayashi ([48], Theorem 5), rephrased for the language of this paper.

Theorem 3 ([48]) *If there are derivation trees π and π' respectively for $\emptyset \vdash M : \mathsf{o}$ and $\emptyset \vdash_{\mathcal{A}} M : q_I$, then there exists a derivation tree π'' for $\emptyset \vdash_{\mathcal{A}} M : q_I$ such that π'' is a refinement of π.*

Let us define the *type size* of a judgment $\Gamma \vdash M : \tau$ by:

$$\begin{array}{l}
\#(\Gamma \vdash M : \tau) = \#\Gamma + \#\tau \\
\#(x_1 : \tau_1, \ldots, x_n : \tau_n) = \#\tau_1 + \cdots + \#\tau_n \\
\#\mathsf{o} = 1 \quad \#(\tau_1 \wedge \cdots \wedge \tau_k \to \tau) = \#\tau_1 + \cdots + \#\tau_n + \#\tau + 1
\end{array}$$

Define the *type width* of a derivation tree for $\Gamma \vdash M : \tau$ as the largest type size of a node of the derivation.

The following theorem follows immediately from Theorem 3 above.

Theorem 4 *Given an automaton $\mathcal{A}$ and a type derivation tree for $\emptyset \vdash M : \mathsf{o}$, $[\![M]\!] \stackrel{?}{\in} \mathcal{L}(\mathcal{A})$ can be decided in time linear in the size of M, under the assumption that the size of $\mathcal{A}$ and the type width of derivation trees are bounded by a constant.*

Proof Due to Theorems 2 and 3, it suffices to check whether there exists a derivation tree π for $\emptyset \vdash_{\mathcal{A}} M : q_I$ that is a refinement of the derivation tree π' for $\emptyset \vdash M : \mathtt{o}$. Since the type width of π' is bounded by a constant, for each subterm N of M, the number of possible judgments that can occur in π is also bounded by a constant (although the constant can be huge). Thus, based on the refinement typing rules, we can enumerate all the valid judgments for N in time linear in the size of N: for example, to enumerate the typing for $M_1 M_2$, first enumerate valid typings for M_1 and M_2 and combine them by using the application rule. Thus, valid typings for M can also be enumerated in time linear in the size of M, and then it suffices to just check whether $\emptyset \vdash_{\mathcal{A}} M : q_I$ is among the valid judgments. □

Remark 5 The complexity is non-elementary if the size of the automaton $\mathcal{A}$ and the type width of derivation trees are not bounded. As in ordinary higher-order model checking [26, 34], if the largest order k of types is fixed, the complexity is k-EXPTIME in the size of the automaton $\mathcal{A}$ and the type width of derivation trees.

The fixed-parameter linear-time algorithm in the proof above is impractical due to the huge constant factor. We can instead use Kobayashi's fixed-parameter linear-time algorithm for higher-order model checking [23]. The algorithm is designed for terms of the *simply-typed* λ-calculus with recursion, represented as a system of top-level function definitions. However, his algorithm can be easily adapted for our language with intersection types: It suffices to convert an input λ-term to a system of toplevel function definitions and then apply Kobayashi's algorithm as it is.

Remark 6 Higher-order model checking [22, 34] usually refers to model checking of the tree generated by a *higher-order recursion scheme* (HORS), which can be considered a functional program. The only differences between HORS and our language are: (i) HORS can be used to describe infinite trees, while our language is only used for describing finite trees, and (ii) HORS must be simply-typed, but our language allows intersection types. Actually, our language can be considered a restriction of the extension of HORS considered by Tsukada and Kobayashi [48]. Because of the restriction to terms generating finite trees, the model checking problem is also closely related to the problem RegLang(r) considered by Terui [45], although he considers the *simply-typed* λ-calculus.

4.2 Data Processing as Program Transformation

In the previous subsection, we considered pattern match queries to answer just yes or no. In practice, it is often required to provide extra information, such as the position of the first match and the number of matching positions. Computation of such extra information can be expressed as tree transducers [7, 10, 11]. The tree transducers defined below are equivalent to the generalized finite state transformations (GFST) introduced by Engelfriet [10] (see Remark 7 below).

Definition 3 (tree transducers) *A tree transducer $\mathcal{X}$ is a quadruple (Σ, Q, q_I, Θ), where Σ is a ranked alphabet, Q is a set of states, q_I is the initial state, and $\Theta\,(\subseteq Q \times \mathit{dom}(\Sigma) \times 2^{\{1,\dots,m\}\times Q} \times \mathit{Terms}_\Sigma)$ (where m is the largest arity of the symbols in Σ) satisfies: if $(q, a, S, M) \in \Theta$, then $(i, q') \in S$ implies $1 \leq i \leq \Sigma(a)$ and $\emptyset \vdash M :$*

$\underbrace{\mathtt{o} \to \cdots \to \mathtt{o}}_{|S|} \to \mathtt{o}$. *The transduction relation* $(q, T) \longrightarrow_{\mathcal{X}} M$ *is defined inductively by the rule:*

$$\frac{\Sigma(a) = n \qquad (q_{i,j}, T_i) \longrightarrow_{\mathcal{X}} M_{i,j} \text{ for each } i \in \{1, \dots, n\}, j \in \{1, \dots, w_j\} \quad (q, a, \{(i, q_{i,j}) \mid 1 \le i \le n, 1 \le j \le w_j\}, M) \in \Theta}{(q, a\, T_1 \cdots T_n) \longrightarrow_{\mathcal{X}} M\, M_{1,1} \cdots M_{1,w_1} \cdots M_{n,1} \cdots M_{n,w_n}}$$

Here, we assume that $q_{i,j} < q_{i,j'}$ *if* $j < j'$ *with respect to the linear order on refinement types (recall Definition 2). We write* $\mathcal{X}(T)$ *for the set of trees* $\{[\![M]\!] \mid (q_I, T) \longrightarrow_{\mathcal{X}} M\}$*, and call an element of* $\mathcal{X}(T)$ *an output of the transducer* $\mathcal{X}$ *for* T*.*

A transducer $\mathcal{X} = (\Sigma, Q, q_I, \Theta)$ *is deterministic if, for each pair* $(q, a) \in Q \times dom(\Sigma)$*, there is at most one* (S, M) *such that* $(q, a, S, M) \in \Theta$*.*

When $\mathcal{X}(T)$ is a singleton set, by abuse of notation, we sometimes write $\mathcal{X}(T)$ for the element of $\mathcal{X}(T)$. Note that, for a deterministic transducer, $\mathcal{X}(T)$ is empty or a singleton set.

Example 12 Let $\Sigma_2 = \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{s} \mapsto 1, \mathtt{e} \mapsto 0\}$. Consider the transducer $\mathcal{X}_1 = (\Sigma_2, \{q_0, q_1\}, q_0, \Theta)$ where Θ is given by:

$$\Theta = \{(q_0, \mathtt{b}, \{(1, q_1)\}, \lambda x.x), (q_0, \mathtt{a}, \{(1, q_0)\}, \mathtt{s}), (q_1, \mathtt{b}, \{(1, q_1)\}, \mathtt{s}), (q_1, \mathtt{a}, \emptyset, \mathtt{e})\}$$

Given a Σ_2-labeled tree T without $\mathtt{s}$, $\mathcal{X}_1$ returns the depth of the first occurrence of a subterm of the form $\mathtt{b}(\mathtt{a}(\cdots))$ in unary representation. For example, for $T = \mathtt{a}(\mathtt{b}(\mathtt{a}(\mathtt{a}(\mathtt{b}(\mathtt{b}(\mathtt{e}))))))$, we have: $(q_0, T) \longrightarrow_{\mathcal{X}} \mathtt{s}((\lambda x.x)\mathtt{e})$. Thus, $\mathcal{X}_1(T) = \{\mathtt{s}(\mathtt{e})\}$. □

Example 13 Let $\Sigma_3 = \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{e} \mapsto 0, \mathtt{br} \mapsto 2\}$. Consider the transducer $\mathcal{X}_2 = (\Sigma_3, \{q_0, q_{odd}, q_{even}\}, q_0, \Theta)$ where Θ is given by:

$$\begin{aligned}\Theta = \{&(q_0, \mathtt{br}, \{(1, q_{odd}), (2, q_{even})\}, \lambda x.\lambda y.\mathtt{br}\, y\, x), (q_0, \mathtt{br}, \{(1, q_{odd}), (2, q_{odd})\}, \mathtt{br}),\\ &(q_0, \mathtt{br}, \{(1, q_{even}), (2, q_{even})\}, \mathtt{br}), (q_0, \mathtt{br}, \{(1, q_{even}), (2, q_{odd})\}, \mathtt{br}),\\ &(q_{odd}, \mathtt{a}, \{(1, q_{even})\}, \mathtt{a}), (q_{even}, \mathtt{a}, \{(1, q_{odd})\}, \mathtt{a}), (q_{even}, \mathtt{e}, \emptyset, \mathtt{e})\}\end{aligned}$$

It takes a tree of the form $\mathtt{br}\ (\mathtt{a}^m(\mathtt{e}))\ (\mathtt{a}^n(\mathtt{e}))$ as an input, and swaps the subtrees $\mathtt{a}^m(\mathtt{e})$ and $\mathtt{a}^n(\mathtt{e})$ only if m is odd and n is even. The transducer is not deterministic, but outputs a singleton set. □

Remark 7 The definition of tree transducers above deviates from the standard definition of top-down tree transducers [7], in that transducers may copy or ignore the result of processing subtrees. For example, consider the transducer $\mathcal{X}_3 = ((q_0, \mathtt{a}, \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 2, \mathtt{e} \mapsto 0\}, \{q_0\}, q_0, \Theta_3)$, where Θ_3 is given by:

$$\Theta = \{(q_0, \mathtt{a}, \{(1, q_0)\}, \lambda x.\mathtt{b}\, x\, x), (q_0, \mathtt{e}, \emptyset, \mathtt{e})\}$$

Then $\mathcal{X}_3$ transforms a unary tree of the form $\mathtt{a}^m\mathtt{e}$ to a complete binary tree of height m, which is not possible for standard top-down tree transducers [7, 10]. As mentioned already, the tree transducers defined above are equivalent to GFST [10], which properly subsume both (ϵ-free) bottom-up and top-down transducers. □

The goal of this subsection is, given a program M and a tree transducer $\mathcal{X}$, to construct a program N that produces an element of $\mathcal{X}([\![M]\!])$. The construction should satisfy the following properties.

P1: It should be reasonably efficient (which also implies N is not too large). In particular, it should be often faster than actually constructing $[\![M]\!]$ and then applying the transducer.

P2: It should be easy to apply further operations (such as pattern matching as discussed in the previous section) on N.

A naive approach to construct N would be to express the transducer $\mathcal{X}$ as a program f, and let N be $f(M)$.[3] This approach obviously does not satisfy the second criterion, however.

We first discuss an approach based on an extension of higher-order model checking in Section 4.2.1, and then discuss an alternative approach based on fusion transformation [13, 44] in Section 4.2.2.

4.2.1 Model Checking Approach

We extend the higher-order model checking discussed in Section 4.1 to compute the output of a transducer (without decompression). Let $\mathcal{X} = (\Sigma, Q, q_I, \Theta)$ be a tree transducer. We shall define a type-directed, non-deterministic transformation relation $\Psi \vdash_{\mathcal{X}} M : \theta \Longrightarrow N$, where θ is a refinement type introduced in Section 4.1. Intuitively, it means that if the value of M is traversed by transducer $\mathcal{X}$ as specified by θ, then the output of the transducer is (the tree or function on trees represented by) N. As a special case, if M represents a tree and if $\Psi \vdash_{\mathcal{X}} M : q_I \Longrightarrow N$, then N is an output of $\mathcal{X}$, i.e., $[\![N]\!] \in \mathcal{X}([\![M]\!])$. The relation $\Psi \vdash_{\mathcal{X}} M : \theta \Longrightarrow N$ is inductively defined by the following rules.

$$\frac{}{\Psi, x : \theta \vdash_{\mathcal{X}} x : \theta \Longrightarrow x_\theta} \quad (\textsc{Tr-Var})$$

$$\frac{(q, a, \{(i, q_{i,j}) \mid 1 \leq i \leq \Sigma(a), 1 \leq j \leq w_i\}, N) \in \Theta}{\Psi \vdash_{\mathcal{X}} a : \bigwedge_{j \in \{1,\ldots,w_1\}} q_{1,j} \to \cdots \to \bigwedge_{j \in \{1,\ldots,w_{\Sigma(a)}\}} q_{\Sigma(a),j} \to q \Longrightarrow N} \quad (\textsc{Tr-Const})$$

$$\frac{\Psi, x : \theta_1, \ldots, x : \theta_n \vdash_{\mathcal{X}} M : \theta \Longrightarrow N \qquad x \text{ does not occur in } \Psi}{\Psi \vdash_{\mathcal{X}} \lambda x.M : \theta_1 \wedge \cdots \wedge \theta_n \to \theta \Longrightarrow \lambda x_{\theta_1}.\cdots\lambda x_{\theta_n}.N} \quad (\textsc{Tr-Abs})$$

$$\frac{\Psi \vdash_{\mathcal{X}} M_1 : \theta_1 \wedge \cdots \wedge \theta_n \to \theta \Rightarrow N_1 \qquad \forall i \in \{1, \ldots, n\}. \Psi \vdash_{\mathcal{X}} M_2 : \theta_i \Rightarrow N_{2,i}}{\Psi \vdash_{\mathcal{X}} M_1 M_2 : \theta \Rightarrow N_1\, N_{2,1}\, \cdots\, N_{2,n}}$$

(Tr-App)

Basically, the transformation works in a compositional manner. Note that if we remove the part "$\Longrightarrow N$", the rules above are essentially the same as the refinement typing rules. In rule Tr-Const, the transformation for constants is determined by the transducer. In rule Tr-App, if the argument M_2 in the original program should have multiple refinement types $\theta_1, \ldots, \theta_n$, we separately translate the argument M_2 for each type and replicate the argument, as the result of the transformation depends on the type of M_2. Thus, in rule Tr-Abs for functions, the function $\lambda x.M$ of type $\theta_1 \wedge \cdots \wedge \theta_n \to \theta$ is transformed into a function that takes n arguments.

[3] Strictly speaking, as our language does not have deconstructors for tree constructors $a_1, \ldots, a_n \in dom(\Sigma)$, we need to transform M into $M'\, a_1\, \cdots\, a_n$ where M' is a pure λ-term, and then transform it into $f\, M'\, a_1\, \cdots\, a_n$.

Example 14 Consider the following program to compute $(\mathtt{ab})^2\mathtt{e}$:

$$\textbf{let } \mathit{twice} = \lambda f.\lambda z.f(f(z)) \textbf{ in } \mathit{twice}\,(\lambda z.\mathtt{a}(\mathtt{b}(z)))\,\mathtt{e}$$

Let us consider the transducer $\mathcal{X}$ given in Example 12. Let ρ and Ψ be $(q_1 \to q_0) \wedge (\top \to q_1) \to \top \to q_0$ and $\mathit{twice} : \rho$ respectively. Then, we have:

$$\begin{array}{l}\Psi \vdash_{\mathcal{X}} \mathit{twice} : \rho \Longrightarrow \mathit{twice}_\rho \\ \Psi \vdash_{\mathcal{X}} \lambda z.\mathtt{a}(\mathtt{b}(z)) : q_1 \to q_0 \Longrightarrow \lambda z_{q_1}.\mathtt{s}((\lambda x.x)z_{q_1}) \\ \Psi \vdash_{\mathcal{X}} \lambda z.\mathtt{a}(\mathtt{b}(z)) : \top \to q_1 \Longrightarrow \mathtt{e}\end{array}$$

Thus, we obtain:

$$\Psi \vdash_{\mathcal{X}} \mathit{twice}(\lambda z.\mathtt{a}(\mathtt{b}(z)))\mathtt{e} : q_0 \Longrightarrow \mathit{twice}_\rho\,(\lambda z_{q_1}.\mathtt{s}((\lambda x.x)z_{q_1}))\,\mathtt{e}$$

The body of *twice* is transformed as follows.

$$\emptyset \vdash_{\mathcal{X}} \lambda f.\lambda z.f(f(z)) : \rho \Longrightarrow \lambda f_{q_1 \to q_0}.\lambda f_{\top \to q_1}.f_{q_1 \to q_0} f_{\top \to q_1}$$

Thus, after some obvious simplifications (such as $(\lambda x.x)M =_\beta M$), we obtain the following program.

$$\textbf{let } \mathit{twice} = \lambda f_1.\lambda f_2.f_1(f_2) \textbf{ in } \mathit{twice}\,(\lambda z.\mathtt{s}\,z)\,\mathtt{e}$$

By evaluating it, we get $(\mathtt{s}\,\mathtt{e})$, which is the output of $\mathcal{X}$ applied to $(\mathtt{ab})^2\mathtt{e}$. $\square$

The following theorem guarantees the correctness of the transformation.

Theorem 5 *Let $\mathcal{X}$ be a tree transducer. If $\emptyset \vdash_{\mathcal{X}} M : q_I \Longrightarrow N$, then $[\![N]\!] \in \mathcal{X}([\![M]\!])$. Conversely, if $\mathcal{X}([\![M]\!])$ is not empty, then there exists N such that $\emptyset \vdash_{\mathcal{X}} M : q_I \Longrightarrow N$ and $[\![N]\!] \in \mathcal{X}([\![M]\!])$.*

We prepare a few lemmas to prove the theorem above. For a transducer $\mathcal{X} = (\Sigma, Q, q_I, \Theta)$, we write $\mathcal{A}_{\mathcal{X}}$ for the automaton (Σ, Q, q_I, Δ) where $\Delta = \{(q, a, S) \mid (q, a, S, N) \in \Theta\}$. We first show that the transformation is a conservative extension of the refinement type system in the following sense.

Lemma 2 *If $\Psi \vdash_{\mathcal{X}} M : \theta \Longrightarrow N$, then $\Psi \vdash_{\mathcal{A}_{\mathcal{X}}} M : \theta$. Conversely, if $\Psi \vdash_{\mathcal{A}_{\mathcal{X}}} M : \theta$, then there exists N such that $\Psi \vdash_{\mathcal{X}} M : \theta \Longrightarrow N$.*

Proof This follows immediately from the fact that each refinement rule is obtained from a transformation rule TR-XX by removing the part "$\Longrightarrow N$". $\square$

Next, we show that the transformation preserves typing. We define the translation $(\cdot)^\flat$ from refinement types (resp., refinement type environments) to simple types (resp. simple type environments) by:

$$\begin{array}{l} q^\flat = \mathtt{o} \\ (\theta_1 \wedge \cdots \wedge \theta_k \to \theta)^\flat = {\theta_1}^\flat \to \cdots \to {\theta_k}^\flat \to \theta^\flat \\ (x_1 : \theta_1, \ldots, x_k : \theta_k)^\flat = x_{1,\theta_1} : {\theta_1}^\flat, \ldots, x_{k,\theta_k} : {\theta_k}^\flat \end{array}$$

Lemma 3 *If $\Psi \vdash_{\mathcal{X}} M : \theta \Longrightarrow N$, then $\Psi^\flat \vdash_{\mathcal{A}_{\mathcal{X}}} N : \theta^\flat$.*

Proof This follows by induction on the derivation of $\Psi \vdash_{\mathcal{X}} M : \theta \Longrightarrow N$, with case analysis on the last rule.

- Case Tr-Var. In this case, $\Psi = \Psi', x : \theta$ with $M = x$ and $N = x_\theta$. Since $\Psi^\flat = {\Psi'}^\flat, x_\theta : \theta^\flat$, we have $\Psi^\flat \vdash x_\theta : \theta^\flat$ as required.
- Case Tr-Const. In this case, $M = a$ with $\theta = \bigwedge_{j\in\{1,\dots,w_1\}} q_{1,j} \to \cdots \to \bigwedge_{j\in\{1,\dots,w_{\Sigma(a)}\}} q_{\Sigma(a),w_{\Sigma(a)}} \to q$ and $(q, a, \{(i, q_{i,j}) \mid 1 \le i \le \Sigma(a), 1 \le j \le w_i\}, N) \in \Theta_{\mathcal{X}}$. By the definition of transducers, we have $\emptyset \vdash N : \mathtt{o}^{w_1+\cdots+w_{\Sigma(a)}} \to \mathtt{o}$. Thus, we have $\Psi^\flat \vdash N : \theta^\flat$ as required.
- Case Tr-Abs. In this case, we have:

$$M = \lambda x.M_1 \qquad N = \lambda x_{\theta_1}.\cdots\lambda x_{\theta_k}.N_1 \qquad \theta = \theta_1 \wedge \cdots \wedge \theta_k \to \theta_0$$
$$\Psi, x : \theta_1, \dots, x : \theta_k \vdash_{\mathcal{X}} M_1 : \theta_0 \Longrightarrow N_1$$

  By the induction hypothesis, we have $\Psi^\flat, x_{\theta_1} : {\theta_1}^\flat, \dots, x_{\theta_k} : {\theta_k}^\flat \vdash N_1 : {\theta_0}^\flat$. By using T-Abs, we get $\Psi^\flat \vdash N : \theta^\flat$ as required.
- Case Tr-App. In this case, we have:

$$M = M_0 M_1 \qquad N = N_0\, N_1 \cdots N_k$$
$$\Psi \vdash_{\mathcal{X}} M_0 : \theta_1 \wedge \cdots \theta_k \to \theta \Longrightarrow N_0$$
$$\Psi \vdash_{\mathcal{X}} M_1 : \theta_i \Longrightarrow N_i \text{ (for each } i \in \{1, \dots, k\})$$

  By the induction hypothesis, we have: $\Psi^\flat \vdash N_0 : {\theta_1}^\flat \to \cdots \to {\theta_k}^\flat \to \theta^\flat$ and $\Psi^\flat \vdash N_i : {\theta_i}^\flat$ for $i \in \{1, \dots, k\}$. By applying Tr-App, we obtain $\Psi^\flat \vdash N : \theta^\flat$ as required.

□

Remark 8 By Lemma 3 above, the output of the transformation is always *simply-typed*: no intersection types are needed. Thus, the output of the transformation may not be sufficiently compressed, and we may wish to apply a post-processing to further compress the output. Recall the term $M_{2,n}$ in Example 1 where $n = 2$:

$$(\lambda f.f\, f\, \mathtt{a}\, \mathtt{c})(\lambda g.\lambda x.g(g\, x)).$$

Consider the identity transducer: $\mathcal{X} = (\{\mathtt{a} \mapsto 1, \mathtt{c} \mapsto 0\}, \{q\}, q, \Theta)$ where:

$$\Theta = \{(q, \mathtt{a}, \{(1, q)\}, \mathtt{a}), (q, \mathtt{c}, \emptyset, \mathtt{c})\}.$$

Then we obtain

$$(\lambda f_{\theta_1}.\lambda f_{\theta_2}.f_{\theta_2}\, f_{\theta_1}\, \mathtt{a}\, \mathtt{c})(\lambda g_{\theta_1}.\lambda x_{\theta_0}.g_{\theta_1}(g_{\theta_1}\, x_{\theta_0}))(\lambda g_{\theta_0}.\lambda x_q.g_{\theta_0}(g_{\theta_0}\, x_q))$$

as the output of the transformation. Here, $\theta_0 = q \to q$, $\theta_1 = \theta_0 \to \theta_0$, and $\theta_2 = \theta_1 \to \theta_1$. By extracting the common term $\lambda g.\lambda x.g(g(x))$ as discussed in Section 3, we obtain:

$$(\lambda f.(\lambda f_{\theta_1}.\lambda f_{\theta_2}.f_{\theta_2}\, f_{\theta_1}\, \mathtt{a}\, \mathtt{c})f\, f)(\lambda g.\lambda x.g(g\, x)).$$

By simplifying the part $(\lambda f_{\theta_1}.\lambda f_{\theta_2}.f_{\theta_2}\, f_{\theta_1}\, \mathtt{a}\, \mathtt{c})f\, f$, we get

$$(\lambda f.f\, f\, \mathtt{a}\, \mathtt{c})(\lambda g.\lambda x.g(g\, x)).$$

Thus we have restored the original term, which is not simply-typed. □

As a corollary of the lemmas above, we obtain:

Corollary 1 *If $\emptyset \vdash_{\mathcal{X}} M : q \Longrightarrow N$, then $\emptyset \vdash M : \mathsf{o}$ and $\emptyset \vdash N : \mathsf{o}$.*

Proof $\emptyset \vdash M : \mathsf{o}$ follows from Lemmas 1 and 2. $\emptyset \vdash N : \mathsf{o}$ follows from Lemma 3. $\square$

Next, we show that substitutions and β-reductions preserve the transformation relation.

Lemma 4 *Suppose that $\Psi, x : \theta_1, \ldots, x : \theta_k \vdash_{\mathcal{X}} M : \theta \Longrightarrow N$ and $\Psi \vdash_{\mathcal{X}} M_0 : \theta_i \Longrightarrow N_i$ for each $i \in \{1, \ldots, k\}$, with $x \notin dom(\Psi)$. Then $\Psi \vdash_{\mathcal{X}} [M_0/x]M_1 : \theta_1 \Longrightarrow [N_1/x_{\theta_1}, \ldots, N_k/x_{\theta_k}]N$.*

Proof The derivation for $\Psi \vdash_{\mathcal{X}} [M_0/x]M_1 : \theta_1 \Longrightarrow [N_1/x_{\theta_1}, \ldots, N_k/x_{\theta_k}]N$ can be obtained from that for $\Psi, x : \theta_1, \ldots, x : \theta_k \vdash_{\mathcal{X}} M : \theta \Longrightarrow N$, by replacing each leaf $\Psi, \Psi', x{:}\theta_1, \ldots, x{:}\theta_k \vdash_{\mathcal{X}} x : \theta_i \Longrightarrow x_{\theta_i}$ of the derivation with $\Psi, \Psi' \vdash_{\mathcal{X}} M_0 : \theta_i \Longrightarrow N_i$. $\square$

Lemma 5 *If $\Psi \vdash_{\mathcal{X}} M : \theta \Longrightarrow N$ and $M \longrightarrow_\beta M'$, then there exists N' such that $N \longrightarrow_\beta^* N'$ and $\Psi \vdash_{\mathcal{X}} M' : \theta \Longrightarrow N'$.*

Proof This follows by induction on the derivation of $M \longrightarrow_\beta M'$. Since the induction steps are trivial, we show only the base case, where $M = (\lambda x.M_1)M_2$ and $M' = [M_2/x]M_1$. Suppose $\Psi \vdash_{\mathcal{X}} M : \theta \Longrightarrow N$. Then, we have:

$$\begin{array}{l} N = (\lambda x_{\theta_1}, \ldots, x_{\theta_k}.N_1)\, N_{2,1} \cdots N_{2,k} \\ \Psi \vdash_{\mathcal{X}} M_2 : \theta_i \Longrightarrow N_{2,i} \text{ for each } i \in \{1, \ldots, k\} \\ \Psi, x : \theta_1, \ldots, x : \theta_k \vdash_{\mathcal{X}} M_1 : \theta \Longrightarrow N_1 \end{array}$$

Let N' be $[N_{2,1}/x_{\theta_1}, \ldots, N_{2,k}/x_{\theta_k}]N_1$. By Lemma 4, we have $\Psi \vdash_{\mathcal{X}} M' : \theta \Longrightarrow N'$. Furthermore, $N \longrightarrow_\beta^* N'$ holds as required. $\square$

The transformation is also preserved by the inverse of substitutions and β-expansions, as stated in Lemmas 6 and 7.

Lemma 6 *If $\Psi \vdash_{\mathcal{X}} [M_2/x]M_1 : \theta \Longrightarrow N$, then $\Psi, x : \theta_1, \ldots, x : \theta_k \vdash_{\mathcal{X}} M_1 : \theta \Longrightarrow N_1$ and $\Psi \vdash_{\mathcal{X}} M_2 : \theta_i \Longrightarrow N_{2,i}$ with $\Psi \vdash_{\mathcal{X}} [M_2/x]M_1 : \theta \Longrightarrow [N_{2,1}/x_{\theta_1}, \ldots, N_{2,k}/x_{\theta_k}]N_1$ for some $N_1, N_{2,1}, \ldots, N_{2,k}, \theta_1, \ldots, \theta_k$ (where k may be 0).*

Proof The condition $\Psi \vdash_{\mathcal{X}} [M_2/x]M_1 : \theta \Longrightarrow [N_{2,1}/x_{\theta_1}, \ldots, N_{2,k}/x_{\theta_k}]N_1$ follows from the other conditions and Lemma 4; so, we show only the other conditions. The proof proceeds by induction on the structure of M_1.

- Case $M_1 = x$. The required result holds for $N_1 = x_\theta$ and $N_{2,1} = N$ with $k = 1$ and $\theta_1 = \theta$.
- Case $M_1 = y (\neq x)$. In this case, $N = y_\theta$ and $y : \theta \in \Psi$. The required result holds for $k = 0$ and $N_1 = N$.
- Case $M_1 = a$. The required result holds for $k = 0$ and $N_1 = N$.
- Case $M_1 = \lambda y.M_3$. In this case, we have:

$$\begin{array}{l} \Psi, y : \theta_1', \ldots, y : \theta_\ell' \vdash_{\mathcal{X}} [M_2/x]M_3 : \theta_0' \Longrightarrow N_3 \\ \theta = \theta_1' \wedge \cdots \wedge \theta_\ell' \to \theta_0' \qquad N = \lambda y_{\theta_1'}. \cdots \lambda y_{\theta_\ell'}.N_3 \end{array}$$

By the induction hypothesis, we have:

$$\begin{array}{l}\Psi, y:\theta'_1,\dots,y:\theta'_\ell, x:\theta_1,\dots,x:\theta_k \vdash_{\mathcal{X}} M_3:\theta'_0 \Longrightarrow N_{3,1}\\ \Psi, y:\theta'_1,\dots,y:\theta'_\ell \vdash_{\mathcal{X}} M_2:\theta_i \Longrightarrow N_{2,i} \text{ (for each } i\in\{1,\dots,k\})\end{array}$$

We may assume without loss of generality that y does not occur in M_2, so that we have $\Psi \vdash_{\mathcal{X}} M_2 : \theta_i \Longrightarrow N_{2,i}$ for each $i \in \{1,\dots,k\}$. Let N_1 be $\lambda y_{\theta'_1}.\cdots\lambda y_{\theta'_\ell}.N_{3,1}$. Then we have $\Psi, x:\theta_1,\dots,x:\theta_k \vdash_{\mathcal{X}} M_1 : \theta \Longrightarrow N_1$ as required.

– Case $M_1 = M_{1,1}M_{1,2}$. In this case, we have:

$$\begin{array}{l}\Psi \vdash_{\mathcal{X}} [M_2/x]M_{1,1} : \theta'_1 \wedge\cdots\theta'_\ell \to \theta \Longrightarrow N'_1\\ \Psi \vdash_{\mathcal{X}} [M_2/x]M_{1,2} : \theta'_j \Longrightarrow N'_{2,j} \text{ (for each } j\in\{1,\dots,\ell\})\end{array}$$

By the induction hypothesis, we have:

$$\begin{array}{l}\Psi, x:\theta_{0,1},\dots,x:\theta_{0,k_0} \vdash_{\mathcal{X}} M_{1,1} : \theta'_1 \wedge\cdots\theta'_\ell \to \theta \Longrightarrow N'_{1,1}\\ \Psi, x:\theta_{j,1},\dots,x:\theta_{j,k_j} \vdash_{\mathcal{X}} M_{1,2} : \theta'_j \Longrightarrow N'_{1,2,j} \text{ (for each } j\in\{1,\dots,\ell\})\\ \Psi \vdash_{\mathcal{X}} M_2 : \theta_{i,j} \Longrightarrow N'_{2,i,j} \text{ (for each } i\in\{0,1,\dots,\ell\}, j\in\{1,\dots,k_i\})\end{array}$$

Let $\{\theta_1,\dots,\theta_k\} = \{\theta_{i,j} \mid i\in\{0,\dots,\ell\}, j\in\{1,\dots,k_i\}\}$. For each $i\in\{1,\dots,k\}$, pick a pair of indices (j_i, j'_i) such that $\theta_i = \theta_{j_i,j'_i}$ and let $N_{2,i}$ be N'_{2,j_i,j'_i}. Let N_1 be $N'_{1,1}\,N'_{1,2,1}\cdots N'_{1,2,\ell}$. Then we have the required result.

□

Remark 9 In the lemma above, note that $N = [N_{2,1}/x_{\theta_1},\dots,N_{2,k}/x_{\theta_k}]N_1$ may not hold if $\mathcal{X}$ is non-deterministic. For example, consider the transducer:

$$\begin{array}{l}\mathcal{X} = (\{\mathtt{a}\mapsto 1, \mathtt{b}\mapsto 1, \mathtt{e}\mapsto 0\}, \{q_0,q_1\}, q_0,\\ \quad\{(q_0,\mathtt{a},\{(1,q_0)\},\mathtt{a}), (q_0,\mathtt{a},\{(1,q_1)\},\mathtt{b}),\\ \quad\;(q_1,\mathtt{b},\{(1,q_0)\},\mathtt{b}), (q_0,\mathtt{b},\{(1,q_0)\},\mathtt{a}), (q_0,\mathtt{e},\emptyset,\mathtt{e})\}).\end{array}$$

Let $M_1 = x(x(\mathtt{e}))$ and $M_2 = \lambda z.\mathtt{a}(\mathtt{b}\,z)$ with $N = (\lambda z_{q_0}.\mathtt{b}(\mathtt{b}\,z_{q_0}))((\lambda z_{q_0}.\mathtt{a}(\mathtt{a}\,z_{q_0}))\mathtt{e})$. Then, we can obtain $\emptyset \vdash_{\mathcal{X}} [M_2/x]M_1 : q_0 \Longrightarrow N$ from: $\emptyset \vdash_{\mathcal{X}} M_2 \Longrightarrow \lambda z_{q_0}.\mathtt{b}(\mathtt{b}\,z_{q_0})$ and $\emptyset \vdash_{\mathcal{X}} M_2 \Longrightarrow \lambda z_{q_0}.\mathtt{a}(\mathtt{a}\,z_{q_0})$. However, the proof above only yields $N_1 = x_{q_0\to q_0}(x_{q_0\to q_0}(\mathtt{e}))$ with $N_{2,1} = \lambda z_{q_0}.\mathtt{a}(\mathtt{a}\,z_{q_0})$ or $N_{2,1} = \lambda z_{q_0}.\mathtt{b}(\mathtt{b}\,z_{q_0})$. Thus, $N \neq [N_{2,1}/x_{q_0\to q_0}]N_1$. This comes from the restriction on the syntax of refinement types, that $\theta_1,\dots,\theta_k$ must be different from each other in $\theta_1\wedge\cdots\wedge\theta_k \to \theta$. This problem can be avoided by removing the restriction (and modifying the rule Tr-Abs to avoid the clash of variable names). □

Lemma 7 *If $\Psi \vdash_{\mathcal{X}} M' : \theta \Longrightarrow N'$ and $M \longrightarrow_\beta M'$, then there exist N and N'' such that $N \longrightarrow^*_\beta N''$ with $\Psi \vdash_{\mathcal{X}} M : \theta \Longrightarrow N$ and $\Psi \vdash_{\mathcal{X}} M' : \theta \Longrightarrow N''$.*

Proof The proof proceeds by induction on the derivation of $M \longrightarrow_\beta M'$. As the induction steps are straightforward, we discuss only the base case, where $M = (\lambda x.M_1)M_2$ and $M' = [M_2/x]M_1$. Suppose $\Psi \vdash_{\mathcal{X}} M' : \theta \Longrightarrow N'$. By Lemma 6, we have $\Psi, x : \theta_1,\dots,x:\theta_k \vdash_{\mathcal{X}} M_1 : \theta \Longrightarrow N_1$ and $\Psi \vdash_{\mathcal{X}} M_2 : \theta_i \Longrightarrow N_{2,i}$ with $\Psi \vdash_{\mathcal{X}} [M_2/x]M_1 : \theta \Longrightarrow [N_{2,1}/x_{\theta_1},\dots,N_{2,k}/x_{\theta_k}]N_1$. Thus, the required result holds for
$N = (\lambda x_{\theta_1}.\cdots\lambda x_{\theta_k}.N_1)N_{2,1}\cdots N_{2,k}$ and $N'' = [N_{2,1}/x_{\theta_1},\dots,N_{2,k}/x_{\theta_k}]N_1$. □

The following lemma states that, for a tree, the transformation computes an output of the transducer.

Lemma 8 *Let T be a Σ-labeled tree. Then, $\emptyset \vdash_{\mathcal{X}} T : q \Longrightarrow N$ if and only if $(q, T) \longrightarrow_{\mathcal{X}} N$.*

Proof This follows by induction on the structure of T. Suppose $T = a\, T_1 \ldots T_k$ with $\Sigma(a) = k$. If $\emptyset \vdash_{\mathcal{X}} T : q \Longrightarrow N$, then we have:

$$\begin{array}{l} N = N_0\, N_{1,1} \cdots N_{1,w_1} \cdots N_{k,1} \cdots N_{k,w_k} \\ (q, a, \{(i, q_{i,j}) \mid i \in \{1, \ldots, k\}, j \in \{1, \ldots, w_i\}\}, N_0) \in \Theta_{\mathcal{X}} \\ \emptyset \vdash_{\mathcal{X}} T_i : q_{i,j} \Longrightarrow N_{i,j} \text{ for each } i \in \{1, \ldots, k\} \text{ and } j \in \{1, \ldots, w_i\}. \end{array}$$

By the induction hypothesis, we have $(q_i, T_{i,j}) \longrightarrow_{\mathcal{X}} N_{i,j}$ for each $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, w_i\}$. Thus, we have $(q, T) \longrightarrow_{\mathcal{X}} N$.

Conversely, suppose $(q, T) \longrightarrow_{\mathcal{X}} N$. Then, we have:

$$\begin{array}{l} N = N_0\, N_{1,1} \cdots N_{1,w_1} \cdots N_{k,1} \cdots N_{k,w_k} \\ (q, a, \{(i, q_{i,j}) \mid i \in \{1, \ldots, k\}, j \in \{1, \ldots, w_i\}\}, N_0) \in \Theta_{\mathcal{X}} \\ (q_{i,j}, T_i) \longrightarrow_{\mathcal{X}} N_{i,j} \text{ for each } i \in \{1, \ldots, k\} \text{ and } j \in \{1, \ldots, w_i\}. \end{array}$$

By the induction hypothesis, we get $\emptyset \vdash_{\mathcal{X}} T_i : q_{i,j} \Longrightarrow N_{i,j}$ for each $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, w_i\}$. Thus, we have $\emptyset \vdash_{\mathcal{X}} T : q \Longrightarrow N$ as required. □

We are now ready to prove Theorem 5.

Proof of Theorem 5. Suppose $\emptyset \vdash_{\mathcal{X}} M : q_I \Longrightarrow N$. By Lemma 1 and Theorem 1, there exists $T (= [\![M]\!])$ such that $M \longrightarrow_{\beta}^{*} T$. By Lemma 5, there exists N' such that $N \longrightarrow_{\beta}^{*} N'$ and $\emptyset \vdash_{\mathcal{X}} T : q_I \Longrightarrow N'$. By Lemma 8, we have $(q_I, T) \longrightarrow_{\mathcal{X}} N'$, which implies $[\![N']\!] \in \mathcal{X}([\![M]\!])$. By the condition $N \longrightarrow_{\beta}^{*} N'$, we have $[\![N]\!] = [\![N']\!] \in \mathcal{X}([\![M]\!])$, as required.

Conversely, suppose that $T \in \mathcal{X}([\![M]\!])$. By the definition of $\mathcal{X}([\![M]\!])$, there exists N' such that $(q_I, [\![M]\!]) \longrightarrow_{\mathcal{X}} N'$. By Lemma 8, we have $\emptyset \vdash_{\mathcal{X}} [\![M]\!] : q_I \Longrightarrow N'$. By Lemma 7, there exists N such that $\emptyset \vdash_{\mathcal{X}} M : q_I \Longrightarrow N$. By the first part of this theorem, we have $[\![N]\!] \in \mathcal{X}([\![M]\!])$ as required. □

Remark 10 Because of the problem mentioned in Remark 9, the second part of the theorem above does not guarantee that every element of $\mathcal{X}([\![M]\!])$ is obtained by the transformation if $\mathcal{X}$ is non-deterministic. This is due to the limitation discussed in Remark 9. □

Remark 11 The transformation above is also applicable to an extension of transducers called *high-level transducers* $(\Sigma, Q, q_I, \Theta, N_1, \ldots, N_\ell)$ [12, 46], where for each $(q, a, S, N) \in \Theta$, N has a higher-order function of type $(\kappa_1 \to \cdots \to \kappa_\ell \to \mathsf{o})^{|S|} \to (\kappa_1 \to \cdots \to \kappa_\ell \to \mathsf{o})$, and $\mathcal{X}(T)$ is defined as $\{[\![N\, N_1 \cdots N_\ell]\!] \mid (q_I, T) \longrightarrow_{\mathcal{X}} M\}$. □

Algorithm. Suppose that a derivation tree for $\emptyset \vdash M : \mathsf{o}$ and a transducer $\mathcal{X} = (\Sigma, Q, q_I, \Theta)$ are given. Thanks to the above theorem, we can reuse the algorithm presented in Section 4.1, to decide whether $\mathcal{X}([\![M]\!])$ is non-empty, and if so, output an element of $\mathcal{X}([\![M]\!])$: Let $\mathcal{A}_{\mathcal{X}}$ be an associated automaton (Σ, Q, q_I, Δ), where $\Delta = \{(q, a, S) \mid (q, a, S, M') \in \Theta\}$. Given a program M that generates a tree, we can first check whether $\emptyset \vdash_{\mathcal{A}_{\mathcal{X}}} M : q_I$ holds. If it does not hold, then $\mathcal{X}([\![M]\!])$ is empty, so we are done. Otherwise, we have a derivation tree for $\emptyset \vdash_{\mathcal{A}_{\mathcal{X}}} M : q_I$, from which we can construct a derivation tree for the program transformation relation: $\emptyset \vdash_{\mathcal{X}} M : q_I \Longrightarrow N$, and output N.

By Theorem 4, the above algorithm runs in time linear in the size of M, under the assumption that the size of $\mathcal{X}$ and the type width of the derivation tree for $\emptyset \vdash M : \mathsf{o}$ is bounded by a constant (though the constant factor can be huge as in higher-order model checking).

4.2.2 Fusion Approach

We discuss another approach, based on the idea of shortcut fusion [13, 44]. Recall that the goal was to construct a program N that produces $\mathcal{X}([\![M]\!])$. Here, we can regard M as a tree generator, and transducer $\mathcal{X}$ as a tree consumer. Thus, by using shortcut fusion, we can construct a program N that computes $\mathcal{X}([\![M]\!])$ without constructing the intermediate data $[\![M]\!]$. For the sake of simplicity, we assume below that the transducer $\mathcal{X} = (\Sigma, Q, q_0, \Theta)$ is deterministic and total, i.e., for each $(q, a) \in Q \times \mathit{dom}(\Sigma)$, there exists exactly one (S, M) such that $(q, a, S, M) \in \Theta$.

A (deterministic and total) transducer $\mathcal{X} = (\Sigma, Q, q_0, \Theta)$ (where $Q = \{q_0, \ldots, q_n\}$) can be viewed as the following homomorphism $h_{\mathcal{X}}$ from o (i.e., the set of Σ-labeled trees) to $Q \to \mathsf{o}$:

$$h_{\mathcal{X}}\ (a\ x_1\ \ldots x_k) = f_a\ (h_{\mathcal{X}}\ x_1)\ \ldots\ (h_{\mathcal{X}}\ x_k) \qquad (\Sigma(a) = k)$$

Here, f_a is given by:

$$\begin{array}{l}
f_a\, g_1\, \cdots\, g_k\, q = \\
\quad \textbf{case}\ q\ \textbf{of} \\
\qquad q_0\ \ \Rightarrow M_0\, (g_1\, q_{0,1,1}) \cdots (g_1\, q_{0,1,w_{0,1}}) \cdots (g_k\, q_{0,k,1}) \cdots (g_k\, q_{0,k,w_{0,k}}) \\
\quad\ \ \mid \cdots \\
\quad\ \ \mid q_n\ \ \Rightarrow M_n\, (g_1\, q_{n,1,1}) \cdots (g_1\, q_{n,1,w_{n,1}}) \cdots (g_k\, q_{n,k,1}) \cdots (g_k\, q_{n,k,w_{n,k}})
\end{array}$$

where $(q_\ell, a, \{(i, q_{\ell,i,j}) \mid 1 \leq i \leq k, 1 \leq j \leq w_{\ell,i}\}, M_\ell) \in \Theta$. By using Church encoding, q_i can be encoded as the function $\lambda q_0. \cdots \lambda q_n.q_i$. Thus, f_a above becomes:

$$\begin{array}{l}
\lambda g_1. \cdots \lambda g_k.\lambda q.q\ (M_0\, (g_1\, q'_{0,1,1}) \cdots (g_1\, q'_{0,1,w_{0,1}}) \cdots (g_k\, q'_{0,k,1}) \cdots (g_k\, q'_{0,k,w_{0,k}})) \cdots \\
\qquad (M_n\, (g_1\, q'_{n,1,1}) \cdots (g_1\, q'_{n,1,w_{n,1}}) \cdots (g_k\, q'_{n,k,1}) \cdots (g_k\, q'_{n,k,w_{n,k}})),
\end{array}$$

of type $\underbrace{\tau_h \to \cdots \to \tau_h}_{k} \to \tau_h$, where $\tau_h = (\underbrace{\mathsf{o} \to \cdots \to \mathsf{o}}_{n+1} \to \mathsf{o}) \to \mathsf{o}$ and $q'_{\ell,i,j} = \lambda q_0. \cdots \lambda q_n.q_{\ell,i,j}$. The following lemma guarantees the correctness of the representation of a transducer as the homomorphism.

Lemma 9 *Let $\mathcal{X} = (\Sigma, Q, q_0, \Theta)$ (where $Q = \{q_0, \ldots, q_n\}$) be a deterministic and total transducer. Then, $[\![h(T)\lambda q_0. \cdots \lambda q_n.q_0]\!] = \mathcal{X}(T)$.*

Proof It suffices to show that, for every tree U, $h_{\mathcal{X}}(T)\lambda q_0. \cdots \lambda q_n.q_\ell \longrightarrow_\beta^* U$ if and only if $(q_\ell, T) \longrightarrow_{\mathcal{X}} \longrightarrow_\beta^* U$. The proof proceeds by induction on the structure of T. Suppose $T = a\, T_1\, \cdots\, T_k$ and $(q_\ell, a, \{(i, q_{i,j}) \mid i \in \{1, \ldots, k\}, j \in \{1, \ldots, w_i\}\}, M) \in \Theta_{\mathcal{X}}$.

To show the "only if" part, assume that $h_{\mathcal{X}}(T)\lambda q_0. \cdots \lambda q_n.q_\ell \longrightarrow_\beta^* U$ for some tree U. Then, by the definition of $h_{\mathcal{X}}$, we have:

$$\begin{array}{l}
h_{\mathcal{X}}(T)\lambda q_0. \cdots \lambda q_n.q_\ell \\
\longrightarrow_\beta^* M\, (h_{\mathcal{X}}(T_1)\lambda q_0. \cdots \lambda q_n.q_{1,1}) \cdots (h_{\mathcal{X}}(T_1)\lambda q_0. \cdots \lambda q_n.q_{1,w_1}) \cdots \\
\qquad (h_{\mathcal{X}}(T_k)\lambda q_0. \cdots \lambda q_n.q_{k,1}) \cdots (h_{\mathcal{X}}(T_k)\lambda q_0. \cdots \lambda q_n.q_{k,w_k}) \\
\longrightarrow_\beta^* U.
\end{array}$$

As $h_{\mathcal{X}}(T_i)\lambda q_0.\cdots\lambda q_n.q_{i,j}$ has type $\mathtt{o}$, by Theorem 1, we have $h_{\mathcal{X}}(T_i)\lambda q_0.\cdots\lambda q_n.q_{i,j} \longrightarrow_\beta^* U_{i,j}$ for some tree $U_{i,j}$ for each $i \in \{1,\ldots,k\}, j \in \{1,\ldots,w_i\}$. By the induction hypothesis, we have $(q_{i,j}, T_i) \longrightarrow_{\mathcal{X}} N_{i,j} \longrightarrow_\beta^* U_{i,j}$. By

$$\begin{array}{l} M\,(h_{\mathcal{X}}(T_1)\lambda q_0.\cdots\lambda q_n.q_{1,1})\cdots(h_{\mathcal{X}}(T_1)\lambda q_0.\cdots\lambda q_n.q_{1,w_1})\cdots \\ \qquad\qquad (h_{\mathcal{X}}(T_k)\lambda q_0.\cdots\lambda q_n.q_{k,1})\cdots(h_{\mathcal{X}}(T_k)\lambda q_0.\cdots\lambda q_n.q_{k,w_k}) \\ \longrightarrow_\beta^* U \end{array}$$

and the confluence of $\longrightarrow_\beta^*$, we have: $M\,U_{1,1}\cdots U_{1,w_1}\cdots U_{k,1}\cdots U_{k,w_k} \longrightarrow_\beta^* U$, which also implies

$$M\,N_{1,1}\cdots N_{1,w_1}\cdots N_{k,1}\cdots N_{k,w_k} \longrightarrow_\beta^* U.$$

Since $(q_\ell, T) \longrightarrow_{\mathcal{X}} M\,N_{1,1}\cdots N_{1,w_1}\cdots N_{k,1}\cdots N_{k,w_k}$, we have $(q_\ell, T) \longrightarrow_{\mathcal{X}}\longrightarrow_\beta^* U$ as required.

To show the "if" part, assume that $(q_\ell, T) \longrightarrow_{\mathcal{X}}\longrightarrow_\beta^* U$. By the definition of $\longrightarrow_{\mathcal{X}}$, we have $(q_{i,j}, T_i) \longrightarrow_{\mathcal{X}} N_{i,j}$ (for each $i \in \{1,\ldots,k\}$ and $j \in \{1,\ldots,w_i\}$) and $M\,N_{1,1}\cdots N_{1,w_1}\cdots N_{k,1}\cdots N_{k,w_k} \longrightarrow_\beta^* U$. By Theorem 1, there exists a tree $U_{i,j}$ such that $N_{i,j} \longrightarrow_\beta^* U_{i,j}$. By the confluence of $\longrightarrow_\beta^*$, we have:

$$M\,U_{1,1}\cdots U_{1,w_1}\cdots U_{k,1}\cdots U_{k,w_k} \longrightarrow_\beta^* U.$$

By the induction hypothesis, we have $h_{\mathcal{X}}(T_i)\lambda q_0.\cdots\lambda q_n.q_{i,j} \longrightarrow_\beta^* U_{i,j}$, so that we obtain:

$$\begin{array}{l} h_{\mathcal{X}}(T)\lambda q_0.\cdots\lambda q_n.q_\ell \\ \longrightarrow_\beta^* M\,(h_{\mathcal{X}}(T_1)\lambda q_0.\cdots\lambda q_n.q_{1,1})\cdots(h_{\mathcal{X}}(T_1)\lambda q_0.\cdots\lambda q_n.q_{1,w_1})\cdots \\ \qquad\qquad (h_{\mathcal{X}}(T_k)\lambda q_0.\cdots\lambda q_n.q_{k,1})\cdots(h_{\mathcal{X}}(T_k)\lambda q_0.\cdots\lambda q_n.q_{k,w_k}) \\ \longrightarrow_\beta^* M\,U_{1,1}\cdots U_{1,w_1}\cdots U_{k,1}\cdots U_{k,w_k} \\ \longrightarrow_\beta^* U \end{array}$$

as required. □

Now, extend $h_{\mathcal{X}}$ to the homomorphism on λ-terms by:

$$\begin{array}{ll} h_{\mathcal{X}}(x) = x & h(a) = f_a \\ h_{\mathcal{X}}(M_1 M_2) = h_{\mathcal{X}}(M_1) h_{\mathcal{X}}(M_2) & h_{\mathcal{X}}(\lambda x.M) = \lambda x.h_{\mathcal{X}}(M) \end{array}$$

h just replaces each tree constructor a with f_a.

The following property follows immediately from the definition.

Lemma 10 *If $M \longrightarrow_\beta N$, then $h_{\mathcal{X}}(M) \longrightarrow_\beta h_{\mathcal{X}}(N)$.*

Proof This follows by induction on the derivation of $M \longrightarrow_\beta N$. As the induction steps are trivial, we discuss only the base case, where $M = (\lambda x.M_1)M_2$ and $N = [M_2/x]M_1$. In this case, we have

$$h_{\mathcal{X}}(M) = (\lambda x.h_{\mathcal{X}}(M_1))h_{\mathcal{X}}(M_2) \longrightarrow_\beta [h_{\mathcal{X}}(M_2)/x]h_{\mathcal{X}}(M_1) = h_{\mathcal{X}}([M_2/x]M_1) = h_{\mathcal{X}}(N)$$

as required. □

As stated in Theorem 6 below, $N = h_{\mathcal{X}}(M)\lambda q_0.\cdots\lambda q_n.q_0$ gives an output of the transducer.

Theorem 6 *Suppose that M is a program of type* $\mathtt{o}$. *Let $\mathcal{X} = (\Sigma, Q, q_0, \Theta)$ be a deterministic and total transducer. Then, we have:*

$$[\![h_{\mathcal{X}}(M)\lambda q_0, \ldots, q_n.q_0]\!] = \mathcal{X}([\![M]\!]).$$

Proof Let $T = [\![M]\!]$, i.e., $M \longrightarrow_{\beta}^{*} T$. Then by using Lemmas 9 and 10, we obtain:

$$\begin{aligned} & N = h_{\mathcal{X}}(M)\lambda q_0.\cdots\lambda q_n.q_0 \\ & \longrightarrow_{\beta}^{*} h_{\mathcal{X}}(T)\lambda q_0.\cdots\lambda q_n.q_0 \\ & \longrightarrow_{\beta}^{*} \mathcal{X}(T) \end{aligned}$$

as required. □

We used syntactic reasoning in the above proof. Alternatively, we can use semantic techniques [13, 44].

We have assumed above that $\mathcal{X}$ is deterministic and total. One way to remove the assumption would be to extend the homomorphism $h_{\mathcal{X}}$ so that it returns a function that maps a state to *a list of* trees, and express the list by using Church encoding. We omit the details since the encoding is tedious and the result of transformation would be too complex for a practical use.

Example 15 Recall Example 14. With the fusion-based approach, we get the following program as the output of transformation:

$$\begin{aligned} & \textbf{let } q_0' = \lambda q_0.\lambda q_1.q_0 \textbf{ in} \\ & \textbf{let } q_1' = \lambda q_0.\lambda q_1.q_1 \textbf{ in} \\ & \textbf{let } f_a = \lambda g.\lambda q.q\,(\mathtt{s}(g\,q_0'))\,\mathtt{e} \textbf{ in} \\ & \textbf{let } f_b = \lambda g.\lambda q.q\,((\lambda x.x)(g\,q_1'))\,(\mathtt{s}(g\,q_1')) \textbf{ in} \\ & \textbf{let } f_e = \lambda q.q\,\bot\,\bot \textbf{ in} \\ & \textbf{let } \mathit{twice} = \lambda f.\lambda z.f(f(z)) \textbf{ in } \mathit{twice}(\lambda z.f_a(f_b(z)))\,f_e\,q_0' \end{aligned}$$

Here, $\bot$ is a special tree constructor denoting an undefined tree, introduced to make $\mathcal{X}$ total. □

There are trade-offs between the two approaches (model checking and fusion approaches). Recall the two properties **P1** and **P2** discussed at the beginning of this section.

- Property **P1** is satisfied by both the model checking and fusion approaches, although the latter is better. The model checking approach runs in time linear in the program size only under the assumption that the type width for the type derivation of the program is fixed. In the worst case, the constant factor can be as large as the size of the uncompressed tree $[\![M]\!]$, in which case the model checking approach is as costly as the naive approach of constructing $[\![M]\!]$ and then applying the transducer. However, thanks to the model checking algorithm, this problem does not always show up, as confirmed by experiments. The fusion approach runs in time linear in the program size unconditionally.
- Property **P2** is better satisfied by the model checking approach. As is clear from Examples 14 and 15, the model checking approach reduces the program more aggressively than the fusion approach, which just postpones the computation involved in the transducer. Furthermore, the fusion approach raises the order of the program (where the order of a program is the largest order of the type of a function) by

two. This can have a very bad effect on further pattern match queries or data manipulations based on higher-order model checking (described in Section 4.1). The model checking approach does not raise the order, although it may raise the arity of functions (e.g., in Remark 8, the unary function $\lambda f.f\ f\ \mathtt{a}\ \mathtt{c}$ has been transformed to the binary function $\lambda f_{\theta_1}.\lambda f_{\theta_2}.f_{\theta_2}\ f_{\theta_1}\ \mathtt{a}\ \mathtt{c}$). Note that for higher-order model checking, the order of programs is the most important factor that affects the worst-case complexity [26, 34, 45].
  Despite the drawback of the fusion approach, we think it is still better (in terms of property **P2**) than the naive approach of expressing the transducer as a program f and just returning $f(M)$, in that the fusion approach avoids the construction of the intermediate tree $[\![M]\!]$ when the output of the transducer needs to be computed.
- The terms generated by the model checking approach is always simply-typed (recall Remark 8), while those generated by the fusion approach may not.

Because of the second points, we think the model checking approach is preferable, and the fusion approach (or the other naive approaches discussed at the beginning of Section 4.1) should be used only when the model checking approach is too slow.

5 Implementation and Experiments

We have implemented the following two prototype systems, which can be tested at `http://www-kb.is.s.u-tokyo.ac.jp/~koba/compress/`.

1. A data compression system based on the algorithm described in Section 3: It takes a tree as an input, and outputs a λ-term that generates the tree. It is based on the algorithm described in Section 3, but it has a few parameters to adjust heuristics: D, N, and W. D is the depth of the search of the algorithm of Figure 3. The system first applies *compressAsTree* up to depth D, and returns up to W smallest terms. The system then repeats this up to N times. (Thus, the total search depth is $N \times D$, but some candidates are dropped due to the width parameter W.)
2. A system to manipulate compressed data: It takes a program M in the form of a higher-order recursion scheme [34] and an automaton $\mathcal{A}$ (or a transducer $\mathcal{X}$, resp.) as input, and answers whether $[\![M]\!]$ is accepted by $\mathcal{A}$ (or outputs a program that generates $\mathcal{X}([\![M]\!])$). We have implemented a new version of a higher-order model checker based on a refinement of Kobayashi's linear-time algorithm [23] (as the previous model checkers [21, 23] are not fast enough for our purpose), and then added a feature to produce the output of a transducer based on the transformation given in Section 4.

The data compression system mentioned above does not scale to large data, since the sub-algorithm used as *compressAsTree*(M) (in Figure 3) checks each pair of subtrees, which costs a time quadratic in the size of M. Thus, we have also implemented another algorithm for *compressAsTree*, which extracts the most frequent pattern, and used it in the experiments in Section 5.1.2.

5.1 Compression

We report experiments on the data compression system. The main purposes of the experiments are: (i) to check whether interesting patterns can be obtained (to confirm the third advantage discussed in Section 1), and (ii) to check whether there is an advantage in terms of the compression ratio. The first and second points are reported in Sections 5.1.1 and 5.1.2 respectively.

5.1.1 Knowledge/Program Discovery

Natural Number The first experiment is for (unary) trees of the form $\mathtt{a}^n(\mathtt{c})$. For $n = 9$ (with parameters $N = 3, D = 1, W = 4$), the output was:

$$\textbf{let } \mathit{thrice} = \lambda f.\lambda x.f(f(f(x))) \textbf{ in } \mathit{thrice}(\mathit{thrice}\,\mathtt{a})\mathtt{c}.$$

For $n = 16$ (with $N = 10, D = 1, W = 4$), the output was:

$$\textbf{let } \mathit{twice} = \lambda f.\lambda x.f(f(x)) \textbf{ in } (\mathit{twice}\,\mathit{twice})\,\mathit{twice}\,\mathtt{a}\,\mathtt{c}.$$

This is $M_{2,3}$ in Example 1.

Here, we have renamed variables with common names such as *twice*. Thus, common functions such as *twice* and *thrice* have been automatically discovered. The part $\mathit{thrice}(\mathit{thrice}\,\mathtt{a})$ also corresponds to the square function for Church numerals, and $(\mathit{twice}\,\mathit{twice})\,\mathit{twice}$ corresponds to the exponential $2^{2^2} = 16$. This indicates that our algorithm can achieve hyper-exponential compression ratio. (In fact, by running our algorithm by hand, we get $65536 = 2^{2^{2^2}}$; recall Example 8.)

Thue-Morse Sequence Thue-Morse Sequence (A010060 in `http://oeis.org/`) t_n is the 0-1 sequence generated by:

$$t_0 = \mathtt{0} \qquad t_n = t_{n-1}s_{n-1}$$

where s_i is the sequence obtained from t_i by interchanging `0`s and `1`s. For example, $t_3 = \mathtt{01101001}$ and $t_4 = \mathtt{0110100110010110}$.

We have encoded a 0-1-sequence into a unary tree consisting of `a` (for 0), `b` (for 1), and `e` (for the end of the sequence): for example, `011` was represented by $\mathtt{a}\,(\mathtt{b}\,(\mathtt{b}\,\mathtt{e}))$. For the 10th sequence t_{10} (with $N = 20, D = 1, W = 4$), the output was:

$$\begin{array}{l}
\textbf{let } \mathit{rep} = \lambda x.\lambda y.\lambda z.x\,(y\,(y\,(x\,z))) \textbf{ in}\\
\textbf{let } \mathit{step} = \lambda f.\lambda a.\lambda b.\mathit{rep}\,(f\,a\,b)\,(f\,b\,a) \textbf{ in}\\
\textbf{let } \mathit{iter} = \mathit{step}\,(\mathit{step}\,(\mathit{step}\,\mathit{rep})) \textbf{ in}\\
\textbf{let } t_8 = \mathit{iter}\,\mathtt{a}\,\mathtt{b} \textbf{ in let } s_8 = \mathit{iter}\,\mathtt{b}\,\mathtt{a} \textbf{ in}\\
\quad t_8\,(s_8\,(s_8\,(t_8\,\mathtt{e})))
\end{array}$$

This is an interesting encoding of the Thue-Morse Sequence. It is known that $t_n = t_{n-2}s_{n-2}s_{n-2}t_{n-2}$ holds for all $n \geq 2$. The above encoding uses this recurrence equation (which has been somehow discovered automatically from only the 10th sequence, not from the definition of Thue-Morse Sequence!), and represents t_{10} as $t_8s_8s_8t_8$. Using the above equation, t_8 and s_8 were represented by $(\mathit{step}^3\ \mathit{rep})\,\mathtt{a}\,\mathtt{b}$ and $(\mathit{step}^3\ \mathit{rep})\,\mathtt{b}\,\mathtt{a}$ respectively.

As for the compression ratio, the length of n-th Thue-Morse Sequence is $O(2^n)$, while the size of the above representation is $O(n)$. For a larger k, the part $\mathit{step}^k\,\mathit{rep}$ (in *iter* above) can further be compressed as in the compression of natural numbers discussed above; thus the hyper-exponential compression ratio is achieved by our algorithm.

Fibonacci Word For the 7th Fibonacci word `abaababaabaababaababa` (with $N = 10, D = 1, W = 4$), one of the outputs was:

$$\begin{array}{l}\textbf{let } f_2 = \lambda y.\mathtt{a}\,(\mathtt{b}\,y) \textbf{ in let } f_3 = \lambda y.f_2\,(\mathtt{a}\,y) \textbf{ in}\\ \textbf{let } f_4 = \lambda y.f_3\,(f_2\,y)) \textbf{ in let } f_5 = \lambda y.f_4\,(f_3\,y) \textbf{ in}\\ \quad f_5\,(f_4\,(f_5\,\mathtt{e}))\end{array}$$

This is almost the definition of Fibonacci word; the last line is equivalent to **let** $f_6 = \lambda y.f_5\,(f_4\,y)$ **in** $f_6\,(f_5\,\mathtt{e})$. (Note again that we have not given the definition of Fibonacci word; we have only given the specific instance.) The system could not, however, find a more compact representation such as the one in Example 6. This is probably due to the limitation discussed at the end of Section 3, that our compression algorithm is not powerful enough to extract some higher-order patterns.

L-system Consider an instance of L-systems, defined by [33]:

$$F_0 = \mathtt{f} \qquad F_{n+1} = F_n[+F_n]F_n[-F_n]F_n$$

where "[", "]", "+", "−" and `f` are terminal symbols. Given the unary tree representation of the sequence F_3 (which is given in Figure 6 of [33]), our system (with $N = 50, D = 1, W = 4$) output the following program in 38 seconds:

$$\begin{array}{l}\textbf{let } \mathit{step} = \lambda g.\lambda z.g(\textbf{let } h = \lambda z.g(](g\,z)) \textbf{ in } [(+(h([-(h\,z)))))\\ \textbf{in } \mathit{step}(\mathit{step}(\mathit{step}(\mathtt{f})))\,\mathtt{e}.\end{array}$$

The function *step* is equivalent to: $\lambda g.\lambda z.g[+g]g[-g]g\,z$, where the applications are treated as right-associative here to avoid too many parentheses. The above output is exactly (a compressed form of) the definition of F_3. Please compare the above result with the following output of Sequitur for F_3 [33]:

S → BFAGA	A → B]B	B→ DFCGC	C → D]D
D → `f`FEGE	E → `f`]`f`	F →[+	G → [−

The output of Sequitur is also compact, but does not tell much about how F_3 has been produced.

English sentences We examined a part of (simple) English text extracted from the article of "Jupiter" in Simple-English version of Wikipedia `http://simple.wikipedia.org/wiki/Jupiter`. The text had 1017 words including punctuations; e.g., "Jupiter's" is considered as 3 words "Jupiter", an apostrophe, and "s". The text was encoded as a unary tree, whose node is labelled by a word instead of a character.

In addition to frequently-appearing phrases such as "million miles away", "in 1979.", and "km/h", interesting higher-order patterns were extracted, such as:

$$\textbf{let } s = \lambda y.\mathtt{APOS}\,(\text{“s”}\,y) \textbf{ in let } \mathit{possessive} = Q\,s \textbf{ in } \ldots$$

The pattern $Q\,s = \lambda n.\lambda y.n(\mathtt{APOS}\,(\text{“s”}\,y))$ expresses the possessive form "A's B". The following combinators B and Q were also extracted.

$$\begin{array}{l}B = \lambda f.\lambda g.\lambda x.f\,(g\,x)\\ Q = \lambda f.\lambda g.\lambda x.g\,(f\,x)\end{array}$$

Bilingual sentences We have also tested our system to compress a sequence of pairs of an English sentence and its French translation by using Google translation (`http://translate.google.com/`). Given an input containing:

$$\texttt{pair}\,(\texttt{I}(\texttt{like}(\texttt{him}(\texttt{period}))))\,(\texttt{Je}(\texttt{le}(\texttt{aime}(\texttt{bien}(\texttt{period})))))$$

and

$$\texttt{pair}\,(\texttt{I}(\texttt{like}(\texttt{her}(\texttt{period}))))\,(\texttt{Je}(\texttt{la}(\texttt{aime}(\texttt{bien}(\texttt{period}))))),$$

our system produced the following output:

$$\begin{array}{l}\textbf{let } x_E = \lambda z.\texttt{pair}\,(\texttt{I}(\texttt{like}(z(\texttt{period})))) \textbf{ in}\\ \textbf{let } x_F = \lambda z.(\texttt{Je}(z(\texttt{aime}(\texttt{bien}(\texttt{period}))))) \textbf{ in}\\ \cdots(x_E\,\texttt{him}\,(x_F(\texttt{le})))\cdots(x_E\,\texttt{her}\,(x_F(\texttt{la})))\cdots\end{array}$$

Thus, the correspondences like “him” vs “le”, “her” vs “la”, and “I like xxx” vs “Je xxx aime bien” have been discovered.

For another example, we have taken 14 simple English sentences from a textbook used in an English course of a kindergarten and fed them and their French translations to our compression system. The word-word or phrase-phrase correspondences that have been found include: “plays with a ...” vs “joue avec un ...”, “friend” vs “ami”, “ball” vs “ballon”, etc. We expect that the reason for the good result is that the English sentences are written for beginners of English language and contain repetitions of simple phrases, so that it is easy to guess the structure of sentences not only for human beings but also for a data compressor.

5.1.2 Compression Size

In this subsection, we report experiments to evaluate the effectiveness of the FPCD approach in terms of the compression size. As mentioned at the beginning of this section, since our naive implementation of the compression algorithm does not scale, we have prepared another implementation, which finds the most frequent tree context and makes it shared at each compression step. We used the following input data, some of which are already used in Section 5.1.1.

- Synthetic data.
  - a4096 = $\texttt{a}^{4096}$.
  - thue-morse-seq11 is the 11th Thue-Morse sequence.
  - fib-word14 (resp. fib-word15) is the 14th (resp. 15th) Fibonacci word.
  - L3 = F_3 and L4 = F_4, where F_n is defined in the instance of L-system considered in Section 5.1.1.
  - cantor-dust is another instance of L-system, consisting of a sequence $A_0, A_1, \ldots, A_6$, where A_i is defined by mutual recursions $A_n = A_{n-1}B_{n-1}A_{n-1}$ and $B_n = B_{n-1}B_{n-1}B_{n-1}$ for $n \geq 1$, with $A_0 = \texttt{a}$ and $B_0 = \texttt{b}$. That is, $A_1 = \texttt{aba}$, $A_2 = \texttt{ababbbaba}$, $A_3 = \texttt{ababbbabababbbbbbbbbababbbaba}$, and so on.
  - square-seq15 (resp. square-seq20) is a sequence of $\texttt{c}(\texttt{a}^{i^2})$ for $i = 1, 2, \ldots, 15$ (resp. for $i = 1, 2, \ldots, 20$).
  - geometric-seq8 is a sequence of $\texttt{c}(\texttt{a}^{2^i})$ for $i = 0, 1, \ldots, 8$.
  - binary-number-gray6 is a sequence of integers 0, 1, ... ,63 represented by 6 bits Gray code. That is, $\texttt{c}[\texttt{aaaaaa}], \texttt{c}[\texttt{aaaaab}], \texttt{c}[\texttt{aaaabb}], \texttt{c}[\texttt{aaaaba}], \ldots, \texttt{c}[\texttt{baaaaa}]$.

name	Original	RE	HORE	HO
a4096	8193	89	28	-
thue-morse-seq11	2049	137	137	53
fib-word14	1221	89	89	83
fib-word15	1975	97	97	87
L3	623	135	135	38
L4	3123	195	195	40
cantor-dust	2201	204	189	147
square-seq15	2541	168	156	-
square-seq20	5821	215	200	-
geometric-seq8	1059	110	101	87
binary-number-gray6	1025	491	491	345
wikipedia-jupiter	2035	1951	1951	-
dna-1000	2001	1087	1084	-
dna-5000	10001	4029	4017	-
dna-10000	20001	7227	7211	-
enwik8-100KB	20235	1362	1346	-
enwik8-200KB	40533	2274	2246	-
enwik8-500KB	101407	4686	4651	-

Table 1 Comparison of the compression size, without/with higher-order patterns.

– DNA sequence: the complete genome of the E. Coli bacterium, taken from the Canterbury Corpus[4]. dna-1000 (resp. dna-5000, dna-10000) is its prefix of length 1000 (resp. 5000, 10000).
– XML Data of Wikipedia, taken from `enwik8`, which is the target data of a compression competition Hutter Prize[5]. As in the experiments for tree compression in [4, 30], we removed `PCData` and attributes, and used the binary-tree encoding. enwik8-100KB (resp. enwik8-200KB, enwik8-500KB) consists of the first 100KB (resp. enwik8-200KB, enwik8-500KB) from enwik8.

The results of the experiments are summarized in Table 1. The first column shows the names of the data, and the second column their original sizes, measured by the size of terms. In the third column "RE", we show the sizes of compressed data, obtained by using *first-order* tree contexts only (thus, essentially equivalent to compressions based on context-free tree grammars [4, 30]). The fourth column "HORE" (Higher-Order REpair) shows the sizes of compressed data, obtained by further compressing the result of the third column by using *higher-order* tree contexts. The last column "HO" shows the result for the naive compression algorithm used in the experiments of Section 5.1.1. As it is too slow, we have not run it for large data. (For L4, the result shown in the column "HO" has been obtained by running the compression algorithm by hand.)

According to the results shown in the columns "RE" and "HORE", except for an extreme case (of a4096), there is no clear evidence that the higher-order compression (using higher-order patterns) is effective in terms of the compression size. We should note however that HORE uses the fixed heuristic for choosing tree contexts (i.e., most frequent contexts), hence it is not taking a full advantage of the higher-order compression. In fact, the naive algorithm, which searches common higher-order contexts more exhaustively, outputs better results, although it takes a significantly longer time.

[4] `http://corpus.canterbury.ac.nz/descriptions/large/E.coli.html`
[5] `http://prize.hutter1.net/`

We plan to test other variations of compression algorithms (e.g. an algorithm that chooses the largest common tree context). We also plan to test the higher-order compression for other variations of data, like music scores and large bilingual texts. For more detailed evaluation of the compression size, we also need to encode the resulting λ-terms into bit strings. It is left for future work.

5.2 Data Processing

We have applied various pattern match queries and transformations to Fibonacci words, to check the scalability of our system with respect to the size of compressed data. Table 2 shows the results. The 2^mth Fibonacci words (for $m = 4, 6, 8, 10, 12, 14$) were represented by using the encoding of Example 6. The size of the representation of the n-th Fibonacci word is $O(\log n)$ (or $O(m)$). The queries and transformations are: Q_1: contains `aa`, Q_2: contains no `bb`, Q_3: contains no `aaa`, T_1: the first occurrence of `aab`, T_2: count the number of `ab`, T_3: replace `ab` with `bb`, TQ_3: T_3 followed by query "contains `bbb`?"). In the row TQ_3, the times do not contain those for applying T_3 (which are shown in the row T_3). All the experiments were conducted on a machine with Intel(R) Xeon(R) CPU with 3 GHz and 8 GB memory.

Our system based on higher-order model checking could quickly answer pattern match queries or apply transformations. The increase of the time with respect to m varies depending on the query or transformation, but an exponential slowdown was not observed for any of the queries and transformations. Note that the length of n-th Fibonacci word is greater than 1.6^{n-1}, so that it is impossible to actually (no matter whether eagerly or lazily) construct the word and then apply a pattern match query or transformation. Even with the grammar-based compression based on context-free grammars, the size of the representation of n-th Fibonacci word is $O(n)$; thus our approach (which runs in time $O(\log n)$) is exponentially faster than the grammar-based approach for this experiment. Our system was relatively slower for TQ_3. This is probably because the transformation T_3 increased the arity of functions, which had a bad effect on model checking. It may be possible to reduce this problem by post-processing the output of the transformation using other program transformation techniques.

It should be noted however that the above result is an extreme case that shows the advantage of our approach. For the real-life data discussed in Section 5.1.2, for which the effect of compression is small, the advantage of compressing data and manipulating them without decompression can be easily offset by the inefficiency of the current higher-order model checker. In order to take advantage of data processing without decompression, we have to wait for further advance of implementation techniques for higher-order model checkers.

6 Related Work

The idea of compressing strings or tree data as functional programs is probably not new; in fact, Tromp [47] studied Kolmogorov complexity in the setting of λ-calculus. We are, however, not aware of any serious previous studies of the approach that propose data compression/manipulation algorithms with a similar capability.

In the context of higher-order model checking, Broadbent et al. ([3], Corollary 3) showed that if t is the tree generated by an order-n higher-order recursion scheme and

Table 2 Times for processing queries and transformations on 2^mth Fibonacci words, measured in seconds.

	m=4	m=6	m=8	m=10	m=12	m=14
Q_1	0.12	0.12	0.12	0.26	0.27	0.27
Q_2	0.03	0.03	0.04	0.12	0.12	0.12
Q_3	0.14	0.14	0.14	0.14	0.14	0.14
T_1	0.13	0.13	0.13	0.27	0.27	0.27
T_2	0.04	0.04	0.12	0.12	0.13	0.13
T_3	0.04	0.04	0.12	0.12	0.13	0.13
TQ_3	0.47	0.62	1.32	1.53	1.84	2.09

$\mathcal{I}$ is a well-formed MSO-interpretation, $\mathcal{I}(t)$ can be generated by an order-$(n+1)$ recursion scheme. As a higher-order recursion scheme can be viewed as a simply-typed λ-term (with recursion) and a transducer can be expressed as a MSO-interpretation, this gives another procedure for the data transformation discussed in Section 4.2 (for the case where the program M is simply-typed). Their transformation is however indirect and quite complex, as it goes through collapsible higher-order pushdown automata [14]. Their transformation also increases the order of the program, as opposed to our transformation given in Section 4.2.1. Thus, we think their transformation is mainly of theoretical interest (indeed, it has never been implemented).

As discussed in Section 2.4, our FPCD approach can be regarded as a generalization of grammar-based compression [1, 4, 30, 33, 36, 38, 39] where a string or a tree is represented as a context-free (string or tree) grammar. Since the problem of computing the smallest CFG that exactly generates w is known to be *NP*-hard [43], various heuristic compression algorithms have been proposed, including Re-pair [27, 30]. Processing of compressed data without decompression has been a hot topic of studies in the grammar-based compression, and our result in Section 4 can be considered a generalization of it to higher-order grammars. In the context of CFG-based compression, however, more operations can be performed without decompression, including the equivalence checking ("given two compressed strings, do they represent the same string?") [35] and compressed pattern matching ("given a compressed string and a *compressed* pattern, does the string match the pattern?") [17]. It is left for future work to investigate whether those operations extend to higher-order grammars.

Charikar et al. [5] introduced the notion of "grammar complexity", which is the size of the smallest context-free grammar that generates a given string. Its advantages over Kolmogorov complexity are that the grammar complexity is computable, and also that there is an efficient algorithm to compute an approximation of the grammar complexity. It would be interesting to consider restrictions of the λ-calculus that subsume context-free grammars but still satisfy such an approximability property.

Our experiment to discover knowledge from the compression of English-French translation, discussed in Section 5.1, appears to be related to studies of example-based machine translation [40], in particular, automatic extraction of translation templates from a bilingual corpus [16]. Nevill-Manning and Witten [33] also report inference of hierarchical (not higher-order, in the sense of the present paper) structures by grammar-based compression.

We have not discussed the issue of how to compactly represent a λ-term (obtained by our compression algorithm) as a bit string. There are a few previous studies to address this issue. Tromp [47] gave two schemes for representing untyped λ-terms as bit strings, one through the de Brujin index representation, and the other through the combinator representation. Vytiniotis and Kennedy [51] introduced a game-based method for representing simply-typed λ-terms as bit-strings.

7 Conclusion

We have studied the approach of compressing data as functional programs, and shown that programming language techniques can be used for compressing and manipulating data. In particular, we have extended a higher-order model checking algorithm to transform compressed data without decompression. The prototype compression and transformation systems have been implemented and interesting experimental results have been obtained.

The work reported in this article should be regarded just as an initial step of studies of the FPCD approach. We plan to address the following issues in future work.

- Theoretical properties of the compression algorithm: As mentioned in Remark 2, the output of our compression algorithm in Section 3 belongs to λ-I calculus. A better characterization is required about the class of λ-terms output by the algorithm.
- Better compression algorithms: The current compression algorithm is not fast enough to be used for large data, and does not exhibit a clear advantage over grammar-based compression in terms of the compression ratio, except for some special cases (recall the results reported in Section 5.1.2). A better compression algorithm is required, which achieves a better balance between the efficiency and the quality of the output.
- Restrictions of the data representation language: Related to the point above, the full λ-calculus may be too powerful for the design of efficient compression algorithms and good theoretical characterizations of them. Thus, it may be worth investigating various restrictions of the λ-calculus. For example, the restriction to terms of order-2 types (recall the definition of the order of types in Remark 1) already subsumes context-free tree grammars.
- Killer applications: The effectiveness of higher-order functions for compression should depend on application domains (natural languages, music data, voice, DNA, etc.). It is currently unclear for what class of data higher-order functions are effective.
- Better higher-order model checking algorithms: We have shown that higher-order model checking can be used manipulating compressed data. The current higher-order model checking algorithms are however not fast enough for manipulating large compressed data.

Acknowledgments We would like to thank Oleg Kiselyov, Tatsunari Nakajima, Kunihiko Sadakane, John Tromp, and anonymous referees for discussions and useful comments. This work was partially supported by Kakenhi 23220001.

References

1. A. Apostolico and S. Lonardi. Some theory and proactive of greedy off-line textual substitution. In *Data Compression Conference 1998 (DCC98)*, pages 119–128, 1998.

2. H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symb. Log.*, 48(4):931–940, 1983.
3. C. H. Broadbent, A. Carayol, C.-H. L. Ong, and O. Serre. Recursion schemes and logical reflection. In *Proceedings of LICS 2010*, pages 120–129. IEEE Computer Society Press, 2010.
4. G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML document trees. *Inf. Syst.*, 33(4-5):456–474, 2008.
5. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.
6. A. Church. *The calculi of lambda-conversion.* Number 6 in Annals of Mathematics Studies. Princeton University Press, 1941.
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. release October, 12th 2007.
8. M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.
9. W. Dam. The IO- and OI-hierarchies. *Theoretical Computer Science*, 20:95–207, 1982.
10. J. Engelfriet. Bottom-up and top-down tree transformations - a comparison. *Mathematical Systems Theory*, 9(3):198–231, 1975.
11. J. Engelfriet, G. Rozenberg, and G. Slutzki. Tree transducers, L systems, and two-way machines. *J. Comput. Syst. Sci.*, 20(2):150–202, 1980.
12. J. Engelfriet and H. Vogler. High level tree transducers and iterated pushdown tree transducers. *Acta Inf.*, 26(1/2):131–192, 1988.
13. A. J. Gill, J. Launchbury, and S. L. Peyton-Jones. A short cut to deforestation. In *FPCA*, pages 223–232, 1993.
14. M. Hague, A. Murawski, C.-H. L. Ong, and O. Serre. Collapsible pushdown automata and recursion schemes. In *Proceedings of LICS 2008*, pages 452–461. IEEE Computer Society, 2008.
15. M. Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability.* Springer-Verlag, Berlin, 2004.
16. H. Kaji, Y. Kida, and Y. Morimoto. Learning translation templates from bilingual text. In *COLING*, pages 672–678, 1992.
17. M. Karpinski, W. Rytter, and A. Shinohara. An efficient pattern matching algorithm for strings with short description. *Nordic Journal on Computing*, 4(2):129–144, 1997.
18. T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theoretical Computer Science*, 1(298):253–272, 2003.
19. T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *Proceedings of FOSSACS 2002*, volume 2303 of *Lecture Notes in Computer Science*, pages 205–222. Springer-Verlag, 2002.
20. N. Kobayashi. Model checking higher-order programs. *Journal of the ACM.* To appear. A revised and extended version of [22] and [21].
21. N. Kobayashi. Model-checking higher-order functions. In *Proceedings of PPDP 2009*, pages 25–36. ACM Press, 2009.
22. N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 416–428, 2009.
23. N. Kobayashi. A practical linear time algorithm for trivial automata model checking of higher-order recursion schemes. In *Proceedings of FOSSACS 2011*, volume 6604 of *Lecture Notes in Computer Science*, pages 260–274. Springer-Verlag, 2011.
24. N. Kobayashi, K. Matsuda, and A. Shinohara. Functional programs as compressed data. In *Proceedings of PEPM 2012*, pages 121–130. ACM Press, 2012.
25. N. Kobayashi and C.-H. L. Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of LICS 2009*, pages 179–188. IEEE Computer Society Press, 2009.
26. N. Kobayashi and C.-H. L. Ong. Complexity of model checking recursion schemes for fragments of the modal mu-calculus. *Logical Methods in Computer Science*, 7(4), 2011.
27. N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. Data Compression Conference '99 (DCC'99)*, pages 296–305. IEEE Computer Society, 1999.
28. M. Li and P. M. B. Vitányi. Kolmogorov complexity and its applications. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 187–254. The MIT Press, 1990.

29. M. Li and P. M. B. Vitányi. *An introduction to Kolmogorov complexity and its applications (3rd ed.)*. Texts in computer science. Springer-Verlag, 2009.
30. M. Lohrey, S. Maneth, and R. Mennicke. Tree structure compression with repair. In *2011 Data Compression Conference (DCC 2011)*, pages 353–362. IEEE Computer Society, 2011.
31. S. Maneth and G. Busatto. Tree transducers and tree compressions. In *Proceedings of FOSSACS 2004*, volume 2987 of *Lecture Notes in Computer Science*, pages 363–377. Springer-Verlag, 2004.
32. W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theoretical Computer Science*, 410(8-10):900–913, 2009.
33. C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *Comput. J.*, 40(2/3):103–116, 1997.
34. C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS 2006*, pages 81–90. IEEE Computer Society Press, 2006.
35. W. Plandowski. Testing equivalence of morphisms on context-free languages. In *ESA'94*, volume 855 of *Lecture Notes in Computer Science*, pages 460–470. Springer-Verlag, 1994.
36. W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.
37. W. Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In *ICALP'04*, volume 3142 of *Lecture Notes in Computer Science*, pages 15–27. Springer-Verlag, 2004.
38. H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3(2-4):416–430, 2005.
39. H. Sakamoto, S. Maruyama, T. Kida, and S. Shimozono. A space-saving approximation algorithm for grammar-based compression. *IEICE Trans. on Information and Systems*, E92-D(2):158–165, 2009.
40. H. L. Somers. Review article: Example-based machine translation. *Machine Translation*, 14(2):113–157, 1999.
41. R. Statman. The typed lambda-calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–81, 1979.
42. C. Stirling. Decidability of higher-order matching. *Logical Methods in Computer Science*, 5(3), 2009.
43. J. Storer. NP-completeness results concerning data compression. Technical Report 234, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N.J., 1997.
44. A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 306–313, New York, NY, USA, 1995. ACM.
45. K. Terui. Semantic evaluation, intersection types and complexity of simply typed lambda calculus. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12)*, volume 15 of *LIPIcs*, pages 323–338. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.
46. A. Tozawa. XML type checking using high-level tree transducer. In *Functional and Logic Programming, 8th International Symposium (FLOPS 2006)*, volume 3945 of *Lecture Notes in Computer Science*, pages 81–96. Springer-Verlag, 2006.
47. J. Tromp. Binary lambda calculus and combinatory logic. In C. S. Claude, editor, *Randomness And Complexity, From Leibniz To Chaitin*, pages 237–260. World Scientific Publishing Company, 2008.
48. T. Tsukada and N. Kobayashi. Untyped recursion schemes and infinite intersection types. In *Proceedings of FOSSACS 2010*, volume 6014 of *Lecture Notes in Computer Science*, pages 343–357. Springer-Verlag, 2010.
49. S. van Bakel. Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 102(1):135–163, 1992.
50. S. van Bakel. Intersection type assignment systems. *Theoretical Computer Science*, 151(2):385–435, 1995.
51. D. Vytiniotis and A. Kennedy. Functional pearl: every bit counts. In *Proceedings of ICFP 2010*, pages 15–26, 2010.

Appendix

A Proofs for Section 2

In this section, we shall prove the following lemma, which corresponds to the "only if" direction of Theorem 1.

Lemma 11 *If $\emptyset \vdash M : \mathsf{o}$, then there exists a tree T such that $M \longrightarrow_\beta^* T$.*

As sketched in the proof of Theorem 1, the lemma follows from more or less standard properties of intersection types [2, 50]. Nevertheless, we provide a self-contained proof of Lemma 11 below to familiarize readers with the type-based transformation techniques used in Section 4.2.

We prove the lemma by using the strong normalization of the simply-typed λ-calculus. To this end, we define a type-based transformation relation $\Gamma \vdash M : \tau \Longrightarrow N$, which transforms a well-typed λ-term M (in the intersection type system) into a corresponding term N of the simply-typed λ-calculus. It is defined by the following extension of typing rules.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \Longrightarrow x_\tau} \quad \text{(STr-Var)}$$

$$\frac{\Sigma(a) = k}{\Gamma \vdash a : \underbrace{\mathsf{o} \to \cdots \to \mathsf{o}}_{k} \to \mathsf{o} \Longrightarrow a} \quad \text{(STr-Const)}$$

$$\frac{\Gamma, x : \tau_1, \ldots, x : \tau_n \vdash M : \tau \Longrightarrow N \qquad x \notin dom(\Gamma) \qquad n \geq 1}{\Gamma \vdash \lambda x.M : \tau_1 \wedge \cdots \wedge \tau_n \to \tau \Longrightarrow \lambda x_{\tau_1}.\cdots \lambda x_{\tau_n}.N} \quad \text{(STr-Abs)}$$

$$\frac{\Gamma \vdash M : \tau \Longrightarrow N \qquad x \notin dom(\Gamma)}{\Gamma \vdash \lambda x.M : \top \to \tau \Longrightarrow \lambda x_\star.N} \quad \text{(STr-AbsT)}$$

$$\frac{\Gamma \vdash M_1 : \tau_1 \wedge \cdots \wedge \tau_n \to \tau \Longrightarrow N_1 \qquad \forall i \in \{1, \ldots, n\}.\Gamma \vdash M_2 : \tau_i \Longrightarrow N_{2,i} \qquad n \geq 1}{\Gamma \vdash M_1 M_2 : \tau \Longrightarrow N_1\, N_{2,1} \cdots N_{2,n}} \quad \text{(STr-App)}$$

$$\frac{\Gamma \vdash M_1 : \top \to \tau \Longrightarrow N_1}{\Gamma \vdash M_1 M_2 : \tau \Longrightarrow N_1\, (\,)} \quad \text{(STr-AppT)}$$

Here, we assume that the simply-typed λ-calculus has a base type $\star (\neq \mathsf{o})$, inhabited by $(\,)$. The idea of the translation to the simply-typed λ-calculus is to replicate each function argument according to its type. Thus, we replicate a formal parameter x to $x_{\tau_1}, \ldots, x_{\tau_n}$ in rule STr-Abs above, and accordingly duplicate an actual parameter M_2 to $N_{2,1}, \ldots, N_{2,n}$ in rule STr-App. The dummy formal (actual, resp.) parameter $x_\star$ ($(\,)$, resp.) is added in STr-AbsT (STr-AppT, resp.) to enforce that the transformation preserves the "shape" of a term. Note that without the dummy parameter, an application $M_1 M_2$ would be transformed to a term of arbitrary shape (when $n = 0$ in STr-AppT).

We first show that the result of the transformation is simply-typed. We write $\Gamma \vdash_{\mathtt{ST}} N : \kappa$ for the standard type judgment relation for the simply-typed λ-calculus, where the syntax of types are generated by

$$\kappa ::= \mathsf{o} \mid \star \mid \kappa_1 \to \kappa_2.$$

We define the translation $\cdot^\dagger$ from intersection types to simple types by:

$$\mathsf{o}^\dagger = \mathsf{o} \qquad (\top \to \tau)^\dagger = \star \to \tau^\dagger \qquad (\tau_1 \wedge \cdots \wedge \tau_k \to \tau)^\dagger = {\tau_1}^\dagger \to \cdots \to {\tau_k}^\dagger \to \tau^\dagger (\text{if } k \geq 1).$$

We extend the translation to type environments by:

$$\Gamma^\dagger = \{x_\tau : \tau^\dagger \mid x : \tau \in \Gamma\}.$$

We assume that $x_\tau = x'_{\tau'}$ if and only if $x = x'$ and $\tau = \tau'$ so that $\Gamma^\dagger$ is a simple type environment.

Lemma 12 *If $\Gamma \vdash M : \kappa \Longrightarrow N$, then $\Gamma^\dagger \vdash_{\mathtt{ST}} N : \kappa^\dagger$.*

Proof This follows by straightforward induction on the derivation of $\Gamma \vdash M : \kappa \Longrightarrow N$. □

Lemma 13 *Suppose $x \notin dom(\Gamma)$. If $\Gamma, x:\tau_1, \dots, x:\tau_k \vdash M : \tau \Longrightarrow N$ and $\Gamma \vdash K : \tau_i \Longrightarrow N_i$ for each $i \in \{1,\dots,k\}$, then $\Gamma \vdash [K/x]M : \tau \Longrightarrow [N_1/x_{\tau_1},\dots,N_k/x_{\tau_k}]N$.*

Proof This follows by induction on the structure of M.

- Case $M = x$: In this case, $N = x_{\tau_i}$ and $\tau = \tau_i$ for some $i \in \{1,\dots,k\}$. The result follows immediately from $\Gamma \vdash K : \tau_i \Longrightarrow N_i$.
- Case $M = y (\neq x)$: In this case, we have $y : \tau \in \Gamma$ and $N = y_\tau$. Thus, $[K/x]M = y$ and $[N_1/x_{\tau_1},\dots,N_k/x_{\tau_k}]N = y_\tau$. By using STr-Var, we have $\Gamma \vdash [K/x]M : \tau \Longrightarrow [N_1/x_{\tau_1},\dots,N_k/x_{\tau_k}]N$ as required.
- Case $M = a$: The result follows immediately, as $M = N = a$ and $\tau = \mathtt{o}^{\Sigma(a)} \to \mathtt{o}$.
- Case $M = \lambda y.M_0$: We can assume that $y \neq x$ without loss of generality. By the assumption, we have:
$$\begin{array}{l}\Gamma, y:\tau'_1,\dots,y:\tau'_\ell \vdash M_0 : \tau' \Longrightarrow N_0\\ (\ell \geq 1 \wedge N = \lambda y_{\tau'_1}.\cdots\lambda y_{\tau'_\ell}.N_0) \vee (\ell = 0 \wedge N = \lambda y_\star.N_0)\\ \tau = \tau'_1 \wedge \cdots \wedge \tau'_\ell \to \tau'\end{array}$$
By the induction hypothesis, we have $\Gamma, y{:}\tau_1,\dots,y{:}\tau_\ell \vdash [K/x]M_0 : \tau' \Longrightarrow [N_1/x_{\tau_1},\dots,N_k/x_{\tau_k}]N_0$. By applying STr-Abs or STr-AbsT, we obtain $\Gamma \vdash [K/x]M : \tau \Longrightarrow N$ as required.
- Case $M = M_1 M_2$: By the assumption, we have:
$$\begin{array}{l}\Gamma \vdash M_1 : \tau'_1 \wedge \cdots \wedge \tau'_\ell \to \tau \Longrightarrow N'_1\\ \Gamma \vdash M_2 : \tau'_i \Longrightarrow N_{2,i} \text{ for each } i \in \{1,\dots,\ell\}\\ (\ell \geq 1 \wedge N = N'_1\, N_{2,1} \cdots N_{2,k}) \vee (\ell = 0 \wedge N = N'_1\,(\,))\end{array}$$
By the induction hypothesis, we have:
$$\begin{array}{l}\Gamma \vdash [K/x]M_1 : \tau_1 \wedge \cdots \wedge \tau_k \to \tau \Longrightarrow [N_1/x_{\tau_1},\dots,N_k/x_{\tau_k}]N'_1\\ \Gamma \vdash [K/x]M_2 : \tau_i \Longrightarrow [N_1/x_{\tau_1},\dots,N_k/x_{\tau_k}]N_{2,i} \text{ for each } i \in \{1,\dots,k\}\end{array}$$
By applying STr-App or STr-AppT, we obtain $\Gamma \vdash [K/x]M : \tau \Longrightarrow [N_1/x_{\tau_1},\dots,N_k/x_{\tau_k}]N$ as required.

□

The following is a special case of Lemma 13 above, where $k = 0$.

Corollary 1 *If $x \notin dom(\Gamma)$ and $\Gamma \vdash M : \tau \Longrightarrow N$, then $\Gamma \vdash [K/x]M : \tau \Longrightarrow N$ holds for any K.*

We are now ready to show the main lemmas (Lemmas 14 and 15 below), which say that if $\Gamma \vdash M : \tau \Longrightarrow N$, then reductions of M and N can be simulated by each other.

Lemma 14 *If $\Gamma \vdash M : \tau \Longrightarrow N$ and $M \longrightarrow_\beta M'$, then there exists N' such that $\Gamma \vdash M' : \tau \Longrightarrow N'$ with $N \longrightarrow_\beta^* N'$.*

Proof This follows by easy induction on the derivation of $\Gamma \vdash M : \tau \Longrightarrow N$. We omit details as the proof is similar to that of Lemma 5 in Section 4.2.1. □

Lemma 15 *If $\Gamma \vdash M : \tau \Longrightarrow N$ and $N \longrightarrow_\beta N'$, then there exist M' and N'' such that $\Gamma \vdash M' : \tau \Longrightarrow N''$ with $M \longrightarrow_\beta^* M'$ and $N' \longrightarrow_\beta^* N''$.*

Proof The proof proceeds by induction on derivation of $\emptyset \vdash M : \tau \Longrightarrow N$, with case analysis on the last rule used. By the assumption $N \longrightarrow_\beta N'$, the last rule cannot be ST-Var or ST-Const.

- Case STr-Abs: In this case, we have:
$$\begin{array}{l}\Gamma, x:\tau_1,\dots,x:\tau_k \vdash M_1 : \tau' \Longrightarrow N_1\\ M = \lambda x.M_1 \qquad N = \lambda x_{\tau_1}.\cdots\lambda x_{\tau_k}.N_1 \qquad N' = \lambda x_{\tau_1}.\cdots\lambda x_{\tau_k}.N'_1\\ N_1 \longrightarrow_\beta N'_1\\ \tau = \tau_1 \wedge \cdots \wedge \tau_k \to \tau'\end{array}$$
By the induction hypothesis, we have M'_1 and N''_1 such that $\Gamma, x:\tau_1,\dots,x:\tau_k \vdash M'_1 : \tau' \Longrightarrow N''_1$ with $M_1 \longrightarrow_\beta^* M'_1$ and $N'_1 \longrightarrow_\beta^* N''_1$. Thus, the required properties hold for $M' = \lambda x.M'_1$ and $N'' = \lambda x_{\tau_1}.\cdots\lambda x_{\tau_k}.N'_1$.

– Case STr-AbsT: Similar to the case STr-Abs.
– Case STr-App: In this case, we have:

$$\begin{array}{l} \Gamma \vdash M_1 : \tau_1 \wedge \cdots \wedge \tau_k \to \tau \Longrightarrow N_1 \quad (k \geq 1) \\ \Gamma \vdash M_2 : \tau_i \Longrightarrow N_{2,i} \text{ for each } i \in \{1, \ldots, k\} \\ M = M_1 M_2 \qquad N = N_1\, N_{2,1} \cdots N_{2,k} \end{array}$$

By the assumption $N \longrightarrow_\beta^* N'$, there are three cases to consider:
(1) $N_1 \longrightarrow_\beta^* N_1'$ with $N' = N_1'\, N_{2,1} \cdots N_{2,k}$.
(2) $N_{2,j} \longrightarrow_\beta^* N_{2,j}$ with $N' = N_1\, N_{2,1} \cdots N_{2,j-1}\, N_{2,j}'\, N_{2,j+1} \cdots N_{2,k}$.
(3) $N_1 = \lambda x.N_1'$ with $N' = ([N_{2,1}/x]N_1')N_{2,2} \cdots N_{2,k}$.
The result for case (1) follows immediately from the induction hypothesis. In case (2), by the induction hypothesis, we have $\Gamma \vdash M_2' : \tau_j \Longrightarrow N_{2,j}''$ with $M_2 \longrightarrow_\beta^* M_2'$ and $N_{2,j}' \longrightarrow_\beta^* N_{2,j}''$ for some M_2' and $N_{2,j}''$. By Lemma 14, for each $i \in \{1, \ldots, k\} \setminus \{j\}$, there exists $N_{2,i}''$ such that $\Gamma \vdash M_2' : \tau_i \Longrightarrow N_{2,i}''$ and $N_{2,i} \longrightarrow_\beta^* N_{2,i}''$. Let $M' = M_1 M_2'$ and $N'' = N_1 N_{2,1}'' \cdots N_{2,k}''$. Then we have $\Gamma \vdash M' : \tau \Longrightarrow N''$ with $M \longrightarrow_\beta^* M'$ and $N' \longrightarrow_\beta^* N''$ as required.
In case (3), by the transformation rules, $\Gamma \vdash M_1 : \tau_1 \wedge \cdots \wedge \tau_k \to \tau \Longrightarrow N_1$ must have been derived from STr-Abs, so that we have:

$$\begin{array}{l} M_1 = \lambda y.M_3 \\ \Gamma, y : \tau_1, \ldots, y : \tau_k \vdash M_3 : \tau \Longrightarrow N_3 \\ (\lambda x.N_1') = (\lambda y_{\tau_1}.\cdots\lambda y_{\tau_k}.N_3) \end{array}$$

Let $M' = [M_2/x]M_3$ and $N'' = [N_{2,1}/y_{\tau_1}, \ldots, N_{2,k}/y_{\tau_k}]N_3$. Then we have $M \longrightarrow_\beta^* M'$ and $N' \longrightarrow_\beta^* N''$. Furthermore, by Lemma 13, we have $\Gamma \vdash M' : \tau \Longrightarrow N''$ as required.
– Case STr-AppT: In this case, we have:

$$\begin{array}{l} \Gamma \vdash M_1 : \top \Longrightarrow N_1 \\ M = M_1 M_2 \qquad N = N_1\,(\,) \end{array}$$

By the assumption $N \longrightarrow_\beta^* N'$, there are two cases to consider:
(1) $N_1 \longrightarrow_\beta^* N_1'$ with $N' = N_1'\,(\,)$.
(2) $N_1 = \lambda x.N_1'$ with $N' = [(\,)/x]N_1'$.
The result for case (1) follows immediately from the induction hypothesis. In case (2), $\Gamma \vdash M_1 : \top\tau \Longrightarrow N_1$ must have been derived from STr-AbsT, so that we have:

$$M_1 = \lambda x.M_3 \qquad \Gamma \vdash M_3 : \tau \Longrightarrow N_1' \qquad x \notin dom(\Gamma)$$

By Lemma 12, we have $\Gamma^\dagger \vdash_{\mathtt{ST}} N_1'$, so that x does not occur in N_1'. Thus, $N' = [(\,)/x]N_1' = N_1'$. Let M' be $[M_2/x]M_3$ and N'' be N'. Then we have $M \longrightarrow_\beta^* M'$ and $N' \longrightarrow_\beta^* N''$. Furthermore, by Lemma 1 we have $\Gamma \vdash M' : \tau \Longrightarrow N''$ as required.

□

Lemma 16 *If $\emptyset \vdash M : \mathtt{o} \Longrightarrow N$ and N is a β-normal form, then M is a tree and $M = N$.*

Proof The proof proceeds by induction on the structure of N. By Lemma 12, we have $\emptyset \vdash N : \mathtt{o}$. Since N does not contain any β-redex, N must be of the form $a\,N_1 \cdots N_k$, where k may be 0. By the transformation rules, we have:

$$\begin{array}{l} M = a\,M_1 \cdots M_k \\ \Sigma(a) = k \\ \emptyset \vdash M_i : \mathtt{o} \Longrightarrow N_i \text{ for each } i \in \{1, \ldots, k\} \end{array}$$

By the induction hypothesis, M_i is a tree and $M_i = N_i$. Thus, M is also a tree and $M = N$ as required. □

We are now ready to prove Lemma 11.

Proof of Lemma 11. If $\emptyset \vdash M : \mathtt{o}$, then by the transformation rules, there exists N such that $\emptyset \vdash M : \mathtt{o} \Longrightarrow N$. By Lemma 12, we have $\emptyset \vdash_{\mathtt{ST}} N : \mathtt{o}$. By the strong normalization property of the simply-typed λ-calculus, there exists a β-normal form N' such that $N \longrightarrow_\beta^* N'$. By Lemma 15, there exists M' such that $M \longrightarrow_\beta^* M'$ and $\emptyset \vdash M' : \mathtt{o} \Longrightarrow N'$. By Lemma 16, M' is a tree. □