Functional Programs as Compressed Data

Naoki Kobayashi
Tohoku University
koba@ecei.tohoku.ac.jp

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

Ayumi Shinohara
Tohoku University
ayumi@ecei.tohoku.ac.jp

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 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 the paper, 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.

Categories and Subject Descriptors E.4 [*Coding and Information Theory*]: Data compaction and compression; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Theory

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, etc., so that decreasing the data size enables more data to be stored in a higher level of the hierarchy, 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 [9].

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 [19, 20], 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 [34]. Despite the generality and potential of the "functional programs as compressed data" 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 "functional programs as compressed data" 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 a (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 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 "abaabaababbbb" 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 [3, 12, 21]. The grammar-based compression has recently been studied actively, and used in practice for compression of XML data [3]. For instance, consider the tree shown in Figure 1 (which has been taken from in [3]). 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:

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

where the sharing of the tree context $\lambda y.\mathtt{c}(A,\mathtt{d}(A,y))$ 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. Thus, as far as data compression is concerned, our approach can be considered a higher-order extension of the grammar-based compression and can achieve a theoretically higher compression ratio.

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,

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

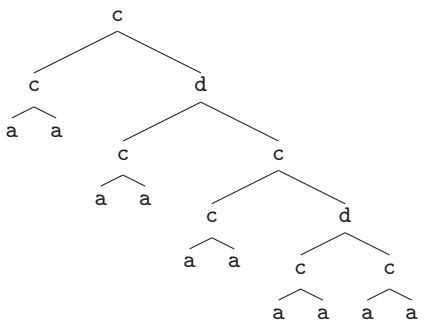

Figure 1. A tree

the main strength of the grammar-based approach [3, 12, 22, 23, 28] is that a large set of operations, such as pattern matching and string replacement, can be performed without decompression. It 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 "functional programs as compressed data" approach also enjoys such a property, by using program verification and transformation techniques. For example, consider a query to ask whether a given tree T matches a given pattern P. It can be rephrased as the query of whether the given tree T is accepted by a tree automaton M_P. Then, the problem of answering the query without decompression is: "Given a functional program p, is the tree generated by p accepted by M_P?". If p is a simply-typed program, then this is just an instance of higher-order model checking problems [14, 15, 25].

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 transducers. Thus, the problem of performing such operations without decompression can be formalized as follows (see also the diagram in Figure 2):

> "Given a tree transducer f and a functional program p that generates a tree T, construct a program q that generates $f(T)$."

Thanks to the "functional programs as compressed data" approach, the construction of the program q is trivial: $q = \hat{f}(p)$, 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, p is a tree generator and $\hat{f}$ is a consumer, so that we can apply the standard fusion transformation [7] to simplify $\hat{f}(p)$. An alternative, more sophisticated approach is, as discussed later, to extend a higher-order model checking algorithm to directly construct q.

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 [9]. As we already discussed, the use of functional programs allows us to compress data to the limit (up to a constant factor), 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. Thus, the equation $3^2 = 9$ and the square function have been automatically discovered by compression.

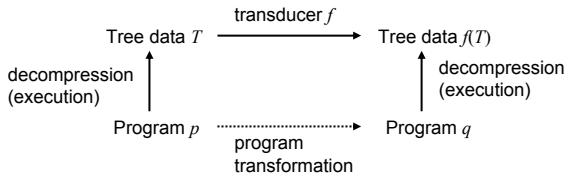

Figure 2. Applying transducers without decompression

In the rest of this paper, 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 paper 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 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).

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 $\mathtt{a}, \mathtt{b}, \ldots$ for elements of the domain of Σ and call them *terminal symbols* (or just symbols). They are used as tree constructors below.

The set $Terms_\Sigma$ 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 ($\mathtt{a}, \mathtt{b}, \ldots$) 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 the 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 $(\lambda x.M_2)M_1$.

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

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

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

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

(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 $[\![M]\!]$ 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 2.1. 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:

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

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

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:

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

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

Example 2.2. Consider the following term, which generates a unary tree $\mathtt{a}^{57}(\mathtt{c})$.

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

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. □

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 2.1 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 $\tau_1 \wedge \cdots \wedge \tau_k \to \tau$. Intuitively, $\mathtt{o}$ 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 $\tau_1 \wedge \cdots \wedge \tau_k \to \tau$.

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 $\{x : \mathtt{o} \to \mathtt{o}, x : (\mathtt{o} \to \mathtt{o}) \to (\mathtt{o} \to \mathtt{o})\}$. We often omit $\{\ \}$ and just write $x_1 : \tau_1, \ldots, x_n : \tau_n$ for $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$.

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:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Sigma(a) = k}{\Gamma \vdash a : \underbrace{\mathtt{o} \to \cdots \to \mathtt{o}}_{k} \to \mathtt{o}}$$

$$\frac{\Gamma, x : \tau_1, \ldots, x : \tau_n \vdash M : \tau \qquad x \text{ does not occur in } \Gamma}{\Gamma \vdash \lambda x.M : \tau_1 \wedge \cdots \wedge \tau_n \to \tau}$$

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

It follows from the standard argument for intersection types [1, 36] that any program that generates a tree is well-typed. (Note that even a term like $(\lambda x.\mathtt{c})\Omega$ that contains the divergent term Ω is typed, by assigning $\top \to \mathtt{o}$ to $\lambda x.\mathtt{c}$.)

Theorem 2.1. *Let M be a λ-term. Then, $\emptyset \vdash M : \mathtt{o}$ if and only if $[\![M]\!]$ is a tree.*

Proof. The "only if" part follows from the facts that (i) $\emptyset \vdash M : \mathtt{o}$ implies that M has a β-normal form, (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. The "if" part follows from the facts that (iv) if M is a tree, then $\emptyset \vdash M : \mathtt{o}$, and (v) typing is preserved by β-expansions (i.e. the inverse of β-reductions). □

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 paper. By using the standard argument for Kolmogorov complexity, we can show that our representation of data as (implicitly-)*typed* λ-terms can be optimal up to an additive constant with respect to Kolmogorov complexity [19, 20]: see [17] for more details. A reader not familiar with Kolmogorov complexity may also wish to consult [19, 20].

Example 2.3. Consider the term:

$$(\lambda \mathit{twice}.\mathit{twice}\ \mathit{twice}\ a\ c)\lambda f.\lambda x.f(f(x)).$$

The part $\lambda \mathit{twice}.\cdots$ is given the following intersection type:

$$\begin{array}{l}(((\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o})\\ \wedge(((\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o}) \to (\mathtt{o} \to \mathtt{o}) \to \mathtt{o} \to \mathtt{o})) \to \mathtt{o}\end{array}$$ □

2.3 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 [3, 12, 21]. 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 $X_j X_k$ where $1 \le j, k < i$. It can be expressed as

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

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. We can also express various extensions of straight-line programs, such as context-free tree grammars [3] and collage systems [12] as λ-terms.

Example 2.4. 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:

$$X_0 = \mathtt{b}, X_1 = \mathtt{a}, X_2 = X_1X_0, \ldots, X_n = X_{n-1}X_{n-2}.$$

It is encoded as:

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

[1] Fibonacci words and its generalization called *Sturmian words* have been studied in a field called *Stringology* [5].

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

Figure 3. Compression Algorithm for λ-terms

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

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

A similar encoding is also possible for an arbitrary number n by using b_0 and b_1 in Example 2.2. □

3. Compression as β-Expansions

This section discusses a compression algorithm which, given a tree T, finds a small λ-term M such that $[\![M]\!] = T$. A naive method would be to enumerate all the λ-terms M smaller than T, and check whether $[\![M]\!] = T$. There are however obvious difficulties. First, the number of terms smaller than T is exponential in the size of T. Secondly, and even worse, it is in general undecidable to check whether $[\![M]\!] = T$. Even if we restrict M to the simply-typed λ-terms, the number of reductions required to compute $[\![M]\!]$ is in general non-elementary [31].

Here, we instead suggest reusing existing algorithms for (context-free) grammar-based tree compression [3, 21], by regarding a λ-term as a term tree (identified up to α-conversion) as follows.

$$x^\sharp = x \quad a^\sharp = a \quad (\lambda x.M)^\sharp = \lambda x \;(\text{child } M^\sharp) \quad (MN)^\sharp = @\;(M^\sharp, N^\sharp)$$

As we have seen in Section 2.3, 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 increase: See Section 5.)

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 [3, 21]. Let us define a context with (up to) k-holes by:

$$C ::= []_1 \mid \cdots \mid []_k \mid x \mid a \mid MC \mid CM \mid \lambda x.C$$

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

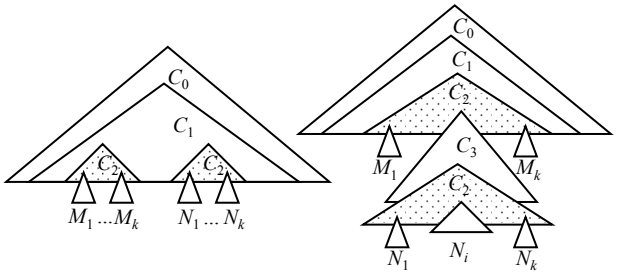

Figure 4. Cases where the common context C_2 occurs horizontally (left, case (i)), and vertically (right, case (ii))

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) $M = C_0[C_1[C_2[M_1, \ldots, M_k]]] \wedge M_i = C_3[C_2[N_1, \ldots, N_k]]$; and
- 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.

Here, (i) and (ii) are the cases where the common context C_2 occurs horizontally and vertically, respectively. 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 $x_1, \ldots, x_k$. This transformation 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 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.

- η-conversion: $\lambda x.M\ x \longrightarrow M$
- β-reduction when the argument is a variable: $(\lambda x.M)y \longrightarrow [y/x]M$
- β-reduction for linear functions: $(\lambda x.M)N \longrightarrow [N/x]M$ where x occurs at most once in M.

Example 3.1. Consider a tree $\mathtt{a}^9(c)$. Let $C_0, C_1, C_2, C_3, M_1, N_1$ be:

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

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:

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

Example 3.2. 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 [22] and also to the regular grammar representation in Figure 1 of [3]. 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 [22] and to the CFG representation in Figure 1 of [3]. □

Relationship with CFG-based Tree Compression Algorithms

As demonstrated in Example 3.2, context-free grammar-based tree compression algorithms [3, 21] 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. [3] 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 [3] 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 3.1 is not possible with CFG-based compression: note that the final term cannot be expressed by a context-free tree grammar.

Limitations

The compression algorithm sketched above is not complete: 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:

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

which consists of a linear tree representing the sequence $\mathtt{a}_1, \ldots, \mathtt{a}_n$ 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

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

is an instance of the higher-order matching problem [32]. Thus, higher-order matching algorithms may be applicable to our data compression scheme, but it is left for future work.

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 $\mathtt{ab}$ as a subpath. Such a pattern matching problem can be formalized as an acceptance problem for tree automata [4].

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

Definition 4.1 (tree automata). A *(top-down, non-deterministic) 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 $\Delta(\subseteq Q \times dom(\Sigma) \times Q^*)$ is a transition function, such that $(q, a, q_1 \cdots q_n) \in \Delta$ implies $\Sigma(a) = n$. The reduction relation $S_1 \longrightarrow S_2$ on subsets of $Q \times \mathcal{T}_\Sigma$ is defined by: $S \cup \{(q, a\,T_1 \cdots T_n)\} \longrightarrow S \cup \{(q_1, T_1), \ldots, (q_n, T_n)\}$ if $(q, a, q_1 \cdots q_n) \in \Delta$. 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 $\mathcal{A}$.

A tree automaton $\mathcal{A} = (\Sigma, Q, q_I, \Delta)$ is *deterministic* if, for each pair $(q, a) \in Q \times dom(\Sigma)$, there is at most one $(q_1, \ldots, q_n)$ such that $(q, a, q_1 \cdots q_n) \in \Delta$.

Example 4.1. Let $\Sigma_1 = \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{e} \mapsto 0\}$. Consider the automaton $\mathcal{A} = (\Sigma_1, \{q_0, q_1, q_f\}, q_0, \Delta)$ where Δ is given by:

$$\begin{array}{c}\Delta = \{(q_0, \mathtt{a}, q_1), (q_0, \mathtt{b}, q_0), (q_1, \mathtt{a}, q_1), (q_1, \mathtt{b}, q_f),\\ (q_f, \mathtt{a}, q_f), (q_f, \mathtt{b}, q_f), (q_f, \mathtt{e}, \epsilon)\}\end{array}$$

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 $\mathcal{A}$. □

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 [15, 35], 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 [13, 25] and the language considered here (see Remark 4.1), we give a direct construction of the type system below.

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 $\theta_1 \wedge \cdots \wedge \theta_k \to \theta$, we require that $\theta_i \neq \theta_j$ if $i \neq j$. Intuitively, q describes the set of trees accepted by M 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.

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:

$$\frac{}{\Psi, x:\theta \vdash_{\mathcal{A}} x:\theta} \qquad \frac{(q, a, q_1 \cdots q_k) \in \Delta}{\Psi \vdash_{\mathcal{A}} a : q_1 \to \cdots \to q_k \to q}$$

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

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

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

Theorem 4.1. *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. This follows from the facts that (i) for any tree T, $T \in \mathcal{L}(\mathcal{A})$ if and only if $\emptyset \vdash_{\mathcal{A}} T : q_I$, and that (ii) the typing is preserved by β-reduction and its inverse. □

Suppose that a derivation for $\emptyset \vdash M : \mathtt{o}$ is given. The result of Tsukada and Kobayashi ([35], 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:

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

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', 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 $\theta_1 :: \tau_1$. The following is the result of Tsukada and Kobayashi ([35], Theorem 5), rephrased for the language of this paper.

Theorem 4.2 ([35]). *If there are derivation trees π and π' respectively for $\emptyset \vdash M : \mathtt{o}$ 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:

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

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 4.2 above.

Theorem 4.3. *Given an automaton $\mathcal{A}$ and a type derivation tree for $\emptyset \vdash M : \mathtt{o}$, $[\![M]\!] \stackrel{?}{\in} \mathcal{L}(\mathcal{A})$ 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 4.1 and 4.2, 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. □

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 [16].[2]

Remark 4.1. Higher-order model checking [15, 25] usually refers to model checking of the tree generated by a *higher-order recursion scheme*, which can be considered a functional program. The only differences between the language of higher-order recursion schemes and our language are: (i) recursion schemes can be used to describe infinite trees, while our language is only used for describing finite trees, and (ii) recursion schemes must be simply-typed, but our language allows intersection types. Actually, our language can be considered a restriction of the extension of recursion schemes considered by Tsukada and Kobayashi [35].

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 [4].

Definition 4.2 (tree transducers). A (top-down, non-deterministic) 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 $\Theta (\subseteq Q \times dom(\Sigma) \times Q^* \times Terms_\Sigma)$ satisfies: if $(q, a, q_1 \cdots q_n, M) \in \Theta$, then $\Sigma(a) = n$ and $\emptyset \vdash M : \underbrace{\mathtt{o} \to \cdots \to \mathtt{o}}_{n} \to \mathtt{o}$. The transduction relation $(q, T) \longrightarrow_{\mathcal{X}} M$ is defined inductively by the rule:

$$\frac{(q_i, T_i) \longrightarrow_{\mathcal{X}} M_i \text{ for each } i \in \{1, \ldots, n\} \qquad (q, a, q_1 \cdots q_n, M) \in \Theta}{(q, a\, T_1 \cdots T_n) \longrightarrow_{\mathcal{X}} M\, M_1 \cdots M_n}$$

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 $(q_1, \ldots, q_n, M)$ such that $(q, a, q_1 \cdots q_n, M) \in \Theta$.

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

Example 4.2. Let $\Sigma_2 = \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{s} \mapsto 1, \mathtt{e} \mapsto 0\}$. Consider the transducer $\mathcal{X} = (\Sigma_2, \{q_0, q_1, q_f\}, q_0, \Theta)$ where Θ is given by:

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

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

[2] Kobayashi's algorithm [16] is for the *simply-typed* λ-calculus with recursion. We can however adapt it for our language with intersection types.

Example 4.3. Let $\Sigma_3 = \{\mathtt{a} \mapsto 1, \mathtt{b} \mapsto 1, \mathtt{e} \mapsto 0, \mathtt{br} \mapsto 2\}$. Consider the transducer $\mathcal{X} = (\Sigma_3, \{q_0, q_{odd}, q_{even}\}, q_0, \Theta)$ where Θ is given by:

$$\Theta = \{(q_0, \mathtt{br}, q_{odd}q_{even}, \lambda x.\lambda y.\mathtt{br}\, y\, x), (q_0, \mathtt{br}, q_{odd}q_{odd}, \mathtt{br}),\\ (q_0, \mathtt{br}, q_{even}q_{even}, \mathtt{br}), (q_0, \mathtt{br}, q_{even}q_{odd}, \mathtt{br}),\\ (q_{odd}, \mathtt{a}, q_{even}, \mathtt{a}), (q_{even}, \mathtt{a}, q_{odd}, \mathtt{a}), (q_{even}, \mathtt{e}, \epsilon, \mathtt{e})\}$$

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. It is a non-deterministic transducer, but outputs a singleton set. □

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 $\mathcal{X}(\llbracket M \rrbracket)$. The construction should satisfy the following properties.

1. It should be reasonably efficient (which also implies N is not too large). In particular, it should be often faster than actually constructing $\llbracket M \rrbracket$ and then applying the transducer.
2. 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 discuss an approach based on an extension of higher-order model checking below. An alternative approach, based on fusion transformation, is discussed in [17].

We can 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., $\llbracket N \rrbracket \in \mathcal{X}(\llbracket M \rrbracket)$. The relation $\Psi \vdash_{\mathcal{X}} M : \theta \Longrightarrow N$ is inductively defined by the following rules.

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

$$\frac{(q, a, q_1 \cdots q_k, N_{q,a,q_1\cdots q_k}) \in \Theta}{\Psi \vdash_{\mathcal{X}} a : q_1 \to \cdots \to q_k \to q \Longrightarrow N_{q,a,q_1\cdots q_k}} \quad \text{(Tr-Const)}$$

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

$$\frac{\Psi \vdash_{\mathcal{X}} M_1 : \theta_1 \wedge \cdots \wedge \theta_n \to \theta \Rightarrow N_1 \quad \forall i \in \{1, \ldots, n\}. \Psi \vdash_{\mathcal{X}} M_2 : \theta_i \Rightarrow N_{2,i}}{\Psi \vdash_{\mathcal{X}} M_1 M_2 : \theta \Rightarrow N_1\, N_{2,1} \cdots N_{2,n}} \quad \text{(Tr-App)}$$

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 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 duplicate the argument. 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 multiple arguments.

Example 4.4. Consider the following program to compute $(\mathtt{ab})^2\mathtt{e}$:

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

Let us consider the transducer $\mathcal{X}$ given in Example 4.2. Let ρ and Ψ be $(q_1 \to q_0) \wedge (q_f \to q_1) \to q_f \to q_0$ and $twice : \rho$ respectively. Then, we have:

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

Thus, we get:

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

The body of *twice* is transformed as follows.

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

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

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

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

The following theorem guarantees the correctness of the transformation. A proof is given in [17].

Theorem 4.4. *Let $\mathcal{X}$ be a tree transducer. If $\emptyset \vdash_{\mathcal{X}} M : q_I \Longrightarrow N$, then $\llbracket N \rrbracket \in \mathcal{X}(\llbracket M \rrbracket)$. Conversely, if $\mathcal{X}(\llbracket M \rrbracket)$ is not empty, then there exists N such that $\emptyset \vdash_{\mathcal{X}} M : q_I \Longrightarrow N$ and $\llbracket N \rrbracket \in \mathcal{X}(\llbracket M \rrbracket)$.*

Remark 4.2. The second part of the theorem above does not guarantee that every element of $\mathcal{X}(\llbracket M \rrbracket)$ is obtained by the transformation.

Remark 4.3. The transformation above is also applicable to an extension of transducers called *high-level transducers* [6].

Algorithm. Suppose that a derivation tree for $\emptyset \vdash M : \mathtt{o}$ and a transducer $\mathcal{X}$ are given. Thanks to the above theorem, we can reuse the algorithm presented in Section 4.1, to decide whether $\mathcal{X}(\llbracket M \rrbracket)$ is non-empty, and if so, output an element of $\mathcal{X}(\llbracket M \rrbracket)$: Let $\mathcal{A}_{\mathcal{X}}$ be an associated automaton (Σ, Q, q_I, Δ), where $\Delta = \{(q, a, q_1 \cdots q_n) \mid (q, a, q_1 \cdots q_n, M') \in \Theta\}$. 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 $\mathcal{X}(\llbracket M \rrbracket)$ 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 4.3, 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 $\emptyset \vdash M : \mathtt{o}$ is bounded by a constant (though the constant factor can be huge as in higher-order model checking).

5. Implementation and Experiments

We have implemented the following two prototype systems, which can be tested at `http://www.kb.ecei.tohoku.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

[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 $f\, M'\, a_1 \cdots a_n$.

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, 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 [25] 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 [16] (as the previous model checkers [14, 16] 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.

5.1 Compression

We report the experimental 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. As the current implementation is naive, scalability is a secondary issue at the moment.

5.1.1 Knowledge/Program Discovery

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

$$\mathbf{let}\ \mathit{thrice} = \lambda f.\lambda x.f(f(f(x)))\ \mathbf{in}\ \mathit{thrice}(\mathit{thrice}\ \mathtt{a})\mathtt{e}.$$

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

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

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 $65536 = 2^{2^{2^2}}$.)

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 0s and 1s. 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:

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

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 `abaababaabaababa ababa` (with $N = 10, D = 1, W = 4$), one of the outputs was:

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

This is almost the definition of Fibonacci word; the last line is equivalent to $\mathbf{let}\ f_6 = \lambda y.f_5\,(f_4\,y)\ \mathbf{in}\ f_6\,(f_5\,\mathtt{e})$. (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 2.4. 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 [24]:

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

where "[", "]", "+", "−" and `f` are terminal symbols. Given the unary tree representation of the sequence F_3 (which is given in Figure 6 of [24]), our system (with $N = 50, D = 1, W = 4$) output the following program in 38 seconds:

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

The function step is equivalent to: $\lambda g.\lambda z.g[+g][-g]g\,z$, 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. The output of Sequitur [24] is more complex.

5.1.2 Compression of Real-Life Data

In this subsection, we report experiments to check whether our compression scheme is also effective (in terms of the compression ratio and whether interesting patterns are discovered) for real-life data. We have tested two kinds of data: XML data and natural language sentences.

To evaluate the effectiveness of compression, we compared the outputs of our compression system with those of grammar-based tree compression tool TreeRePair [21] (`http://code.google.com/p/treerepair/`). The sizes of compressed data were similar for the experiments below, and we could not observe a clear advantage over TreeRePair.[4] Potential reasons for the lack of clear advantages are: (i) Natural language sentences may not have much redundancy, (ii) the data in the experiments are too small to benefit from extraction of higher-order structures, and (iii) our compression algorithm is not good enough to extract complex higher-order structures (recall Section 3).

Our system could, however, find some interesting higher-order patterns, as reported below.

XML Data of Wikipedia We tested the structure of the first 11280 lines (1% in lines) of `enwik8`, which is the target data of Hutter

[4] We omit size data here, since for a fair comparison, we need to encode both λ-terms and grammars into bit strings. Note however that if we first use TreeRePair before applying our system, the result cannot be worse than that of TreeRePair.

Prize[5], a compression competition. As in the experiments for tree compression in [3, 21], we removed `PCData` and attributes and used the binary-tree encoding.

In the output of our system, the higher-order pattern *twice* was extracted to compress the repetition such as `<namespace/>` ... `<namespace/>` appearing in the tree. In addition, the following flipped version of the combinator B ($\lambda f.\lambda g.\lambda x.f\,(g\,x)$) was extracted.

$$Q \quad = \quad \lambda f.\lambda g.\lambda x.g\,(f\,x)$$

English Text 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:

$$\mathbf{let}\ s = \lambda y.\mathtt{APOS}\ (\text{"s"}\ y)\ \mathbf{in}\ \mathbf{let}\ \mathit{possessive} = Q\ s\ \mathbf{in}\ \ldots$$

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

English-French Translation We have also tested our system to compress a sequence of pairs of an English sentence and its French translation. We have taken simple English sentences from a textbook used in an English course of a kindergarten and prepared their French translations by using Google translation (`http://translate.google.com/`). Given an input containing:

`pair(I(like(him(period))))(Je(le(aime(bien(period)))))`

and

`pair(I(like(her(period))))(Je(la(aime(bien(period)))))`,

our system produced the following output:

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

Thus, the correspondences like "him" vs "le", "her" vs "la", and "I like xxx" vs "Je xxx aime bien" have been discovered. Other word-word or phrase-phrase correspondences that have been found from a sequence of 14 pairs of sentences include: "plays with a ..." vs "joue avec un ...", "friend" vs "ami", "ball" vs "ballon", etc.

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 1 shows the results. The 2^mth Fibonacci words (for $m = 4, 6, 8, 10, 12, 14$) were represented by using the encoding of Example 2.4. 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 3Ghz and 8GB 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.

It should be noted however that the above result is an extreme case that shows the advantage of our approach. When the effect of compression is small as in the experiments in Section 5.1.2, the advantage of compressing data and manipulating them without decompression can be easily offset by the inefficiency of the current higher-order model checker.

	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

Table 1. Times for processing queries and transformations on 2^mth Fibonacci words, measured in seconds.

6. Related Work

The idea of compressing strings or tree data as functional programs is probably not new; in fact, Tromp [34] 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. ([2], Corollary 3) showed that if t is the tree generated by an order-n higher-order recursion scheme and $\mathcal{I}$ 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 via collapsible higher-order pushdown automata [8]. Their transformation also increases the order of the program, as opposed to our transformation given in Section 4. Thus, we think their transformation is mainly of theoretical interest (indeed, it has never been implemented).

As discussed in Section 2.3, our approach to compress data as functional programs can be regarded as a generalization of grammar-based compression [3, 21, 24, 27, 29], Since the problem of computing the smallest CFG that exactly generates w is known to be *NP*-hard [33], various heuristic compression algorithms have been proposed, including Re-pair [18, 21]. 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?") [26] and compressed pattern matching ("given a compressed string and a *compressed* pattern, does the string match the pattern?") [11]. It is left for future work to investigate whether those operations extend to higher-order grammars.

[5] `http://prize.hutter1.net/`

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 [30], in particular, automatic extraction of translation templates from a bilingual corpus [10]. Nevill-Manning and Witten [24] also report inference of hierarchical (not higher-order, in the sense of the present paper) structures by grammar-based compression.

We have not discussed how to compactly represent a λ-term (obtained by our compression algorithm) as a bit string. Vytiniotis and Kennedy [37] 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. Larger experiments are left for future work.

Acknowledgments We would like to thank Tatsunari Nakajima, Kunihiko Sadakane, Kazuya Yaguchi, and anonymous referees for discussions and comments. This work was partially supported by Kakenhi 23220001.

References

[1] 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.

[2] 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.

[3] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML document trees. *Inf. Syst.*, 33(4-5):456–474, 2008.

[4] 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.

[5] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2002.

[6] J. Engelfriet and H. Vogler. High level tree transducers and iterated pushdown tree transducers. *Acta Inf.*, 26(1/2):131–192, 1988.

[7] A. J. Gill, J. Launchbury, and S. L. Peyton-Jones. A short cut to deforestation. In *FPCA*, pages 223–232, 1993.

[8] 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.

[9] M. Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. Springer-Verlag, Berlin, 2004.

[10] H. Kaji, Y. Kida, and Y. Morimoto. Learning translation templates from bilingual text. In *COLING*, pages 672–678, 1992.

[11] 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.

[12] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theor. Comput. Sci.*, 1(298):253–272, 2003.

[13] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *Proceedings of FOSSACS 2002*, volume 2303 of *LNCS*, pages 205–222. Springer-Verlag, 2002.

[14] N. Kobayashi. Model-checking higher-order functions. In *Proceedings of PPDP 2009*, pages 25–36. ACM Press, 2009.

[15] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proc. of POPL*, pages 416–428, 2009.

[16] 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 *LNCS*, pages 260–274. Springer-Verlag, 2011.

[17] N. Kobayashi, K. Matsuda, and A. Shonohara. Functional programs as compressed data. An extended version, available from the first author's web page, 2012.

[18] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. Data Compression Conference '99 (DCC'99)*, page 296. IEEE Computer Society, 1999.

[19] 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.

[20] M. Li and P. M. B. Vitányi. *An introduction to Kolmogorov complexity and its applications (3rd ed.)*. Texts in computer science. Springer-Verlag, 2009.

[21] 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.

[22] S. Maneth and G. Busatto. Tree transducers and tree compressions. In *Proceedings of FOSSACS 2004*, volume 2987 of *LNCS*, pages 363–377. Springer-Verlag, 2004.

[23] W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.*, 410(8-10):900–913, 2009.

[24] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *Comput. J.*, 40(2/3):103–116, 1997.

[25] 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.

[26] W. Plandowski. Testing equivalence of morphisms on context-free languages. In *ESA'94*, volume 855 of *LNCS*, pages 460–470. Springer-Verlag, 1994.

[27] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

[28] W. Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In *ICALP'04*, volume 3142 of *LNCS*, pages 15–27. Springer-Verlag, 2004.

[29] H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *Journal of Discrete Algorithms*, 3(2-4):416–430, 2005.

[30] H. L. Somers. Review article: Example-based machine translation. *Machine Translation*, 14(2):113–157, 1999.

[31] R. Statman. The typed lambda-calculus is not elementary recursive. *Theor. Comput. Sci.*, 9:73–81, 1979.

[32] C. Stirling. Decidability of higher-order matching. *Logical Methods in Computer Science*, 5(3), 2009.

[33] J. Storer. NP-completeness results concerning data compression. Technical Report 234, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, N.J., 1997.

[34] J. Tromp. Binary lambda calculus and combinatory logic. In *Kolmogorov Complexity and Applications*, volume 06051 of *Dagstuhl Seminar Proceedings*, 2006.

[35] T. Tsukada and N. Kobayashi. Untyped recursion schemes and infinite intersection types. In *Proceedings of FOSSACS 2010*, volume 6014 of *LNCS*, pages 343–357. Springer-Verlag, 2010.

[36] S. van Bakel. Intersection type assignment systems. *Theor. Comput. Sci.*, 151(2):385–435, 1995.

[37] D. Vytiniotis and A. Kennedy. Functional pearl: every bit counts. In *Proceedings of ICFP 2010*, pages 15–26, 2010.