Compact Bit Encoding Schemes for Simply-Typed Lambda-Terms

Kotaro Takeda Naoki Kobayashi
The University of Tokyo, Japan
{takeda,koba}@kb.is.s.u-tokyo.ac.jp

Kazuya Yaguchi Ayumi Shinohara
Tohoku University, Japan
{kazuya_yaguchi,ayumi}@shino.ecei.tohoku.ac.jp

Abstract

We consider the problem of how to compactly encode simply-typed λ-terms into bit strings. The work has been motivated by Kobayashi et al.'s recent work on higher-order data compression, where data are encoded as functional programs (or, λ-terms) that generate them. To exploit its good compression power, the compression scheme has to come with a method for compactly encoding the λ-terms into bit strings. To this end, we propose two type-based bit-encoding schemes; the first one encodes a λ-term into a sequence of symbols by using type information, and then applies arithmetic coding to convert the sequence to a bit string. The second one is more sophisticated; we prepare a context-free grammar (CFG) that describes only well-typed terms, and then use a variation of arithmetic coding specialized for the CFG. We have implemented both schemes and confirmed that they often output more compact codes than previous bit encoding schemes for λ-terms.

Categories and Subject Descriptors E.4 [*CODING AND INFORMATION THEORY*]

Keywords Simply-typed λ-calculus, Bit encoding, Data compression

1. Introduction

In this paper, we address the issue of how to compactly encode simply-typed λ-terms into bit strings. Our study is motivated by the recent work of Kobayashi et al. on higher-order data compression [6], where tree data are compressed as functional programs (or, λ-terms) that generate them. The advantages of higher-order compression are:

(i) An extremely high compression ratio can be achieved *in theory*. For example, $a^{2^n} b$ can be expressed by $\mathbf{let}\ twice = \lambda f.\lambda x.f(fx)\ \mathbf{in}\ twice^n\, a\, b$. In fact, if the untyped λ-calculus is used, it follows from the standard argument on Kolmogorov complexity that we can achieve an optimal compression ratio up to an additive constant [6].

(ii) Compressed data can be manipulated without decompression, like grammar-based data compression (where data are expressed as context-free grammars that generate them) [1, 7, 9].

ICFP'16, September 18–24, 2016, Nara, Japan
© 2016 ACM. 978-1-4503-4219-3/16/09...$15.00
http://dx.doi.org/10.1145/2951913.2951918

(iii) Many other compression schemes can be easily emulated. For example, the run-length encoding $(ab)^3(abc)^2$ of the string $abababcabc$ can be expressed by:

$$\mathbf{let}\ f_{ab} = \lambda x.a(b\,x)\ \mathbf{in}\ \mathbf{let}\ f_{abc} = \lambda x.a(b(c\,x))\ \mathbf{in}\ \mathbf{let}\ thrice = \lambda x.f(f(f\,x))\ \mathbf{in}\ \mathbf{let}\ twice = \lambda x.f(f\,x)\ \mathbf{in}\ thrice\ f_{ab}(twice\ f_{abc}\ e).$$

Here, e is used to represent the end of the string.

To exploit the high compression power in practice, however, the scheme has to come with a good method to compactly encode a λ-term (obtained as the result of compression) further into a bit string. In fact, the advantage (iii) above can be exploited only if the overhead incurred by the emulation is kept minimal. For the example of the run-length encoding above, the run-length representation is just a sequence "ab 3 abc 2", but the corresponding λ-term $thrice\ f_{ab}(twice\ f_{abc}\ e)$ is tree-structured, and thus a naive encoding would waste bits to represent the structure of the tree.

A good bit-encoding scheme for λ-terms may be useful also for compact representation of source programs written in a functional language, or their compiler intermediate representations, and for run-time serialization of λ-terms, e.g., for the purpose of code migration.

An obvious way to encode a λ-term into a bit string would be to take a string representation of the λ-term (e.g., the string "λ f . λ x . f (f x)" for the term $\lambda f.\lambda x.f(fx)$) and apply an existing string compression scheme to the string. Such a scheme is far from optimal, since the string compression scheme is not aware of the structure of λ-terms and assigns bit code also to (the string representations of) ill-formed terms, like "f λ . λ x . f (f x)".

To our knowledge, there are not so many studies on bit encoding of λ-terms. Tromp [12] proposed a bit encoding scheme for untyped λ-terms in de Bruijn notation [3]. His encoding is not so compact as he uses fixed tag bits for representing whether each subterm is a variable, an abstraction, or an application; this is probably due to the difference of the motivation from ours; his bit encoding was designed so that bit-encoded λ-terms can be easily manipulated by λ-terms. Vytiniotis and Kennedy [5, 13] proposed a game-based bit encoding scheme for simply-typed λ-terms. They encode a λ-term as a sequence of Yes/No answers to questions for identifying the λ-term. In both schemes [5, 12, 13], every bit sequence is guaranteed to be a prefix of some valid λ-term. The obtained bit sequence is, however, not short enough (at least when used in the context of higher-order compression, as confirmed by the experiments reported later).

In the present paper, we propose two compact bit encoding schemes for simply-typed λ-terms: tagless coding and grammar-based coding. In both schemes, we use type information and arithmetic coding [14] for compact bit representations of λ-terms. In the tagless coding, a λ-term is first converted to a sequence of symbols, as in the obvious, naive method discussed above. How-

ever, we use type information effectively to avoid redundant symbols like "(" and ")". For example, consider a term $a(b\,c)$ where $a, b{:}\mathsf{o} \to \mathsf{o}$ and $c{:}\mathsf{o}$. It can be expressed as a sequence "$a\,b\,c$", without parentheses or tag bits to tell whether the first argument of a is the symbol b or the application $b\,c$. When decoding the sequence (given the same type information above), we can safely recover the original term $a(b\,c)$, since the other candidate $(a\,b)\,c$ is ill-typed. For λ-abstractions, it suffices to annotate each λ with the type of the λ-abstraction (instead of the type of the bound variable). Thus, $\lambda x:\mathsf{o} \to \mathsf{o} \to \mathsf{o}.\lambda y:\mathsf{o}.((x\,y)\,y)$, whose de Bruijn representation is $\lambda:\mathsf{o} \to \mathsf{o} \to \mathsf{o}.\lambda:\mathsf{o}.((1\,0)0)$, can be represented by:

$$\lambda_{(\mathsf{o}\to\mathsf{o}\to\mathsf{o})\to\mathsf{o}\to\mathsf{o}}\,\lambda_{\mathsf{o}\to\mathsf{o}}\,1\,0\,0,$$

without any extra tags or parentheses to tell how this sequence should be parsed. The original de Bruijn representation of the term can be safely recovered using the type information. By looking at the first symbol $\lambda_{(\mathsf{o}\to\mathsf{o}\to\mathsf{o})\to\mathsf{o}\to\mathsf{o}}$, we find that the body should have type $\mathsf{o} \to \mathsf{o}$, and by looking at the next symbol $\lambda_{\mathsf{o}\to\mathsf{o}}$, we find that the body is the λ-abstraction (if it were an application, the second λ should be annotated with a type of the form $\tau_1 \to \cdots \to \tau_n \to \mathsf{o} \to \mathsf{o}$ where $n > 0$), and that the rest of the sequence should be parsed as a term of type o, so that $1\,0\,0$ should be parsed as $(1\,0)\,0$. After obtaining a sequence of symbols, we apply arithmetic coding [14]. According to experiments, the overall encoding tends to produce much more compact code than the previous methods [5, 12, 13].

The second scheme we propose in this paper, i.e., grammar-based coding, aims to improve the tagless coding, to produce even more compact code. A defect of the tagless coding explained above is that the sequence of symbols obtained from a λ-term is not uniform (the probability distribution of occurrences of each symbol depends on each position in a sequence) and that information is not properly communicated to the backend arithmetic coder. To overcome the defect, the second scheme integrates the two phases. In order to reflect type information, we first prepare a context-free grammar (CFG) for describing well-typed terms; given finite sets of types and variables that may be used in terms, the grammar can be obtained based on typing rules. For example, if the set of types and variables that may be used in a term are just $\{\mathsf{o}, \mathsf{o} \to \mathsf{o}, \mathsf{o} \to \mathsf{o} \to \mathsf{o}\}$ and $\{0_{\mathsf{o}}, 1_{\mathsf{o}}, 2_{\mathsf{o}\to\mathsf{o}\to\mathsf{o}}\}$ (here we use numbers as variables since we work on de Bruijn representations), then the grammar is:

$$\begin{array}{l} M_{\mathsf{o}} \to 0_{\mathsf{o}} \mid 1_{\mathsf{o}} \mid M_{\mathsf{o}\to\mathsf{o}} M_{\mathsf{o}} \\ M_{\mathsf{o}\to\mathsf{o}} \to \lambda.M_{\mathsf{o}} \mid M_{\mathsf{o}\to\mathsf{o}\to\mathsf{o}} M_{\mathsf{o}} \\ M_{\mathsf{o}\to\mathsf{o}\to\mathsf{o}} \to 2_{\mathsf{o}\to\mathsf{o}\to\mathsf{o}} \mid \lambda.M_{\mathsf{o}\to\mathsf{o}} \end{array}$$

Based on the grammar, a λ-term may be encoded as a sequence of production rules used in the leftmost derivation of the term. For example, the term $\lambda.((2_{\mathsf{o}\to\mathsf{o}\to\mathsf{o}} 0_{\mathsf{o}}) 1_{\mathsf{o}})$ of type $\mathsf{o} \to \mathsf{o}$ would be expressed by:

$$[M_{\mathsf{o}\to\mathsf{o}} \to \lambda.M_{\mathsf{o}}]\,[M_{\mathsf{o}} \to M_{\mathsf{o}\to\mathsf{o}} M_{\mathsf{o}}]\,[M_{\mathsf{o}\to\mathsf{o}} \to M_{\mathsf{o}\to\mathsf{o}\to\mathsf{o}} M_{\mathsf{o}}]$$
$$[M_{\mathsf{o}\to\mathsf{o}\to\mathsf{o}} \to 2_{\mathsf{o}\to\mathsf{o}\to\mathsf{o}}][M_{\mathsf{o}} \to 0_{\mathsf{o}}][M_{\mathsf{o}} \to 1_{\mathsf{o}}]$$

We then apply (a variant of) arithmetic coding for the sequence. The advantage over the first scheme is that, at each position of the sequence, we know production rules of which non-terminal may occur, so that we can take such contextual information into account during the arithmetic coding. In fact, once we get the context-free grammar above, we can simply apply the variation of arithmetic coding adapted for CFG [2]. An additional advantage of the second scheme is that it is easy to reflect some assumptions on the syntax of λ-terms. For example, one may wish to assume that all the λ-terms to be encoded are in A-normal form [10], and avoid allocating bit codes to terms not in A-normal form. That assumption can be reflected in the grammar, resulting in even more compact coding.

We have implemented the two bit encoding schemes above. As the underlying arithmetic coding, we have used range encoding [8], a faster variation of arithmetic coding, and adjusted it to take into account the CFG of simply-typed λ-terms for the second encoding scheme. We have confirmed through experiments that our encoding schemes almost always outperform (in terms of the size of produced bit code) the previous methods [5, 12]. We have also compared our schemes with the bit-encoding module of TreeRePair [7], a grammar-based compression tool. TreeRePair compresses tree data in the form of a straight-line context-free (SLCF) tree grammar (a deterministic context-free tree grammar without recursion), and then encodes the grammar into a bit sequence. Since an SLCF tree grammar can be expressed in a λ-term of the form:

$$\mathbf{let}\ x_1 = M_1\ \mathbf{in}\ \cdots \mathbf{let}\ x_n = M_n\ \mathbf{in}\ N$$

where only first-order functions may occur in $M_1, \ldots, M_n, N$, our bit encoding scheme is applicable to SLCF tree grammars as well. A question is how much we can reduce the overhead of considering general λ-terms; in theory, the restriction to SLCF tree grammars allows more compact coding. Our experimental results show that the second encoding scheme of ours is almost as good as the bit encoder of TreeRePair, despite that the latter is specialized for SLCF tree grammars.

The remainder of this paper is structured as follows. Section 2 defines the goal and briefly reviews the idea of arithmetic coding. Sections 3 and 4 introduce our first and second encoding schemes respectively. Section 5 reports experimental results. Section 6 discusses related work, and Section 7 concludes the paper. Basic knowledge on the simply-typed λ-calculus is assumed in the paper. Knowledge of arithmetic coding [14] and its variants [2, 8] may help but is not prerequisite for reading this paper; we will provide minimal information about them.

2. Preliminaries

In this section, we introduce the syntax of the simply-typed λ-calculus in de Bruijn notation, and set our goal of the paper. We also very briefly explain arithmetic coding, which may be useful for understanding the following sections.

2.1 Simply-Typed λ-Calculus in de Bruijn Notation

The set of types we consider here is given by:

$$\tau ::= \mathsf{o} \mid \tau_1 \to \tau_2.$$

We have the single base type o; in the context of higher-order compression [6], it is used to describe the type of tree data. As usual, $\tau_1 \to \tau_2$ describes functions from τ_1 to τ_2. Our bit encoding schemes, however, do not depend on the fact there is only a single base type. They also work in the presence of other type constructors, such as algebraic data types.

We work over λ-terms in de Bruijn notation [3]:

$$M ::= n \mid \lambda:\tau.M \mid M_1 M_2.$$

Here, n is a natural number, representing de Bruijn index. An index describes the number of λ-abstractions between the occurrence of a variable and that of the binder. For example, the λ-term $\lambda x.\lambda y.y\,x$ in the usual notation is represented by $\lambda.\lambda.0\,1$. For the sake of simplicity, we do not consider constants; they are treated just as free variables, which are also represented by indices. When the names of constants are important in bit encoding, we can separately prepare a list of constant names and encode it using an off-the-shelf string compression tool. As usual, we use the convention that the application constructor is left-associative, and binds tighter than λ, so that $\lambda.0\,1\,1$ represents $\lambda.((0\,1)\,1)$. In examples, we sometimes use the ordinary syntax of λ-terms for readability.

A type judgment for λ-terms is of the form: $\Gamma \vdash M : \tau$, where Γ, called a *type environment*, is a *sequence* of types. The judgment means that M has type τ under the assumption that each variable n

has type $\Gamma(n)$, where $\Gamma(n)$ is the n-th[1] element of Γ. Typing rules are given by:

$$\frac{\Gamma(n) = \tau}{\Gamma \vdash n : \tau}$$

$$\frac{\tau \cdot \Gamma \vdash M : \tau'}{\Gamma \vdash \lambda : \tau . M : \tau \to \tau'}$$

$$\frac{\Gamma \vdash M : \tau' \to \tau \qquad \Gamma \vdash N : \tau'}{\Gamma \vdash MN : \tau}$$

Let $\mathcal{J}$ be the set of triples (Γ, M, τ) (or just pairs (Γ, M), since τ is uniquely determined by Γ and M) such that $\Gamma \vdash M : \tau$. We write $A \to B$ ($A \rightharpoonup B$, resp.) for the sets of total (partial, resp.) functions from A to B. The goal in the present paper is to come up with a pair of (computable) functions $(Enc, Dec) \in (\mathcal{J} \to \{0,1\}^*) \times (\{0,1\}^* \rightharpoonup \mathcal{J})$ such that $Dec(Enc(\Gamma, M, \tau)) = (\Gamma, M, \tau)$ for every $(\Gamma, M, \tau) \in \mathcal{J}$. Furthermore, we request that the length of the bit sequence $Enc(\Gamma, M, \tau)$ tends to be "small".

REMARK 2.1. *The set of λ-terms can also be generated by the following alternative but equivalent inductive definition:*

$$M ::= n\, M_1 \cdots M_k \mid (\lambda : \tau . M_0) M_1 \cdots M_k$$

where $k \geq 0$. The corresponding typing rules are given by:

$$\frac{\Gamma(n) = \tau_1 \to \cdots \tau_k \to \tau \qquad \Gamma \vdash M_i : \tau_i \text{ for each } i \in \{1, \ldots, k\}}{\Gamma \vdash n\, M_1 \cdots M_k : \tau}$$

$$\frac{\tau' \cdot \Gamma \vdash M_0 : \tau'' \qquad \Gamma \vdash M_i : \tau_i \text{ for each } i \in \{1, \ldots, k\} \qquad \tau' \to \tau'' = \tau_1 \to \cdots \to \tau_k \to \tau}{\Gamma \vdash (\lambda : \tau' . M_0) M_1 \cdots M_k : \tau}$$

We often use the above syntax and typing rules in the rest of the paper.

2.2 Arithmetic Coding

A reader may wish to skip this subsection for the first reading, and come back to this subsection when necessary. Arithmetic coding [14] is a scheme for encoding a string into a bit sequence. Like Huffman coding, it takes into account the frequency of occurrences of each letter. Unlike Huffman coding, however, the bit code allocated for each letter may depend on a context, and the code length may be *fractional*.

In Huffman coding, if the probabilities that a, b, and c may occur are $0.5, 0.25$ and 0.25 respectively, then bit codes $0, 10$, and 11 are allocated to them respectively. In this case, the bit length for each letter is equivalent to the entropy of the letter, i.e., $-\log p$ where p is the probability that the letter occurs. A defect of Huffman coding is that it can allocate only bits of integer length. For example, even if the probabilities of a, b, and c are $0.6, 0.2$ and 0.2, we still have no choice but to assign $0, 10$, and 11, whose lengths deviate from the entropies: $-\log 0.6\,(\approx 0.74)$, $-\log 0.2\,(\approx 2.32)$, and $-\log 0.2$.

In arithmetic coding, an interval $[x, y)$ (where $0 \leq x < y \leq 1$) is allocated for each *string*, where the size of the interval $y - x$ represents the probability that the string occurs (given the length). The string is then represented as a prefix of a number in the interval that is long enough to identify the interval. For example, if the probabilities of a, b, and c are $0.6, 0.2$ and 0.2, then the intervals allocated for strings of length 2 are:

$aa : [0, 0.36)$	$ab : [0.36, 0.48)$	$ac : [0.48, 0.6)$
$ba : [0.6, 0.72)$	$bb : [0.72, 0.76)$	$bc : [0.76, 0.8)$
$ca : [0.9, 0.92)$	$cb : [0.92, 0.96)$	$cc : [0.96, 1)$

Since any number $0.00\cdots$ in the binary representation belongs to $[0, 0.25) \subseteq [0, 0.36)$, aa may be represented by 00 (by omitting the part "0."). Similarly, the string ac may be represented by 1000, since any number $0.1000\cdots$ in the binary representation belongs to $[0.48, 0.6)$. It is known that as the length of an input string increases, the length of the bit code approaches to the entropy of the string.

The basic arithmetic coding requires a separate table describing the probability of each letter; in *adaptive* arithmetic coding, it is avoided by dynamically calculating the probability of each letter from the number of occurrences of the letter in the sequence encoded/decoded so far. In the implementation, we use a variant of (adaptive) arithmetic coding, called (adaptive) range coding [8], where integer intervals $[m, n)$ (with $0 \leq m < n \leq M$ for a sufficiently large M) are used instead of floating point intervals.

3. First Approach: Type-based Tagless Coding of λ-Terms

This section describes our first encoding scheme: tagless coding. As explained in Section 1, we first convert a λ-term to a sequence of symbols, consisting of variables (i.e., de Bruijn indices) and λ_τ. By utilizing type information, we can omit tags to distinguish between variables, applications, and λ-applications. We then use a string compressor (a range coder [8], in our implementation), to further convert the sequence to a bit sequence. Sections 3.1 and 3.2 below describe the first and second steps respectively.

3.1 Encoding/Decoding of λ-Terms into/from Sequences

As already explained, each λ-term is encoded into a sequence consisting of indices and symbols of the form λ_τ. We first explain the idea through examples. Consider a term $0\,(0\,1\,1)\,(0\,1\,1)$, which has type $\mathtt{o}$ under the type environment $(\mathtt{o} \to \mathtt{o} \to \mathtt{o}) \cdot \mathtt{o}$ (recall that a type environment is expressed as a sequence of types; it represents $0 : \mathtt{o} \to \mathtt{o} \to \mathtt{o}, 1 : \mathtt{o}$ in the ordinary notation). The term is encoded to a sequence:

$$0\,0\,1\,1\,0\,1\,1,$$

obtained by just listing up the variables occurring in the term from left to right. Given the information that the sequence represents a term that has type $\mathtt{o}$ under $(\mathtt{o} \to \mathtt{o} \to \mathtt{o}) \cdot \mathtt{o}$, we can safely recover the original term as follows. First, since the head of the sequence is 0, we know that the sequence represents a term of the form $0\,M_1 \cdots M_k$ (where k may be 0). Since 0 has type $\mathtt{o} \to \mathtt{o} \to \mathtt{o}$, and the term should have type $\mathtt{o}$, we know that the term is actually of the form $0\,M_1\,M_2$, where both M_1 and M_2 should have type $\mathtt{o}$; thus, the remaining sequence: "$0\,1\,1\,0\,1\,1$" consists of two substrings, which represent M_1 and M_2 respectively. Now, since the head is 0, we know that M_1 is of the form $0\,M_{1,1}\,M_{1,2}$ where both $M_{1,1}$ and $M_{1,2}$ have type $\mathtt{o}$, and that $M_{1,1}$ is encoded by a prefix of the remaining sequence "$1\,1\,0\,1\,1$". Since the head is 1 and 1 has type $\mathtt{o}$, $M_{1,1}$ must be the variable 1. Now, we know $M_{1,2}$ is encoded by a prefix of the remaining sequence "$1\,0\,1\,1$". By applying the same reasoning, we know $M_{1,2}$ is also 1. By repeating the same reasoning for M_2, we can recover the original term $0\,(0\,1\,1)\,(0\,1\,1)$.

As another example, let us consider the term:

$$(\lambda : \mathtt{o} \to \mathtt{o} \to \mathtt{o} . \lambda : \mathtt{o} . (1\,0)\,0)0,$$

[1] Counting from 0.

which has type $\mathtt{o} \to \mathtt{o}$ under the type environment $\mathtt{o} \to \mathtt{o} \to \mathtt{o}$ (In the ordinary notation, the term is expressed as $(\lambda x : \mathtt{o} \to \mathtt{o} \to \mathtt{o}.\lambda y : \mathtt{o}.(x\, y)\, y)z$, which has type $\mathtt{o} \to \mathtt{o}$ under $z : \mathtt{o} \to \mathtt{o} \to \mathtt{o}$.) The term is encoded into the sequence:

$$\lambda_{(\mathtt{o}\to\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}}\, \lambda_{\mathtt{o}\to\mathtt{o}}\, 1\,0\,0\,0,$$

obtained again by just listing up λ's and variables occurring in the term. Note, however, that each λ is annotated with the type of the λ-abstraction, not that of the bound variable; see Remark 3.1 below. Given the sequence above along with the type environment $\mathtt{o} \to \mathtt{o} \to \mathtt{o}$ and the type of the term $\mathtt{o} \to \mathtt{o}$, we can recover the original term as follows. Since the head of the symbol is $\lambda_{(\mathtt{o}\to\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}}$, we know that the term must be of the form $(\lambda.M_0)M_1 \cdots M_k$. By the type information, we also know that $k = 1$, so that the term is actually of the form $(\lambda : \mathtt{o} \to \mathtt{o} \to \mathtt{o}.M_0)M_1$, where M_0 and M_1 have types $\mathtt{o} \to \mathtt{o}$ and $\mathtt{o} \to \mathtt{o} \to \mathtt{o}$ under the environments $(\mathtt{o} \to \mathtt{o} \to \mathtt{o}) \cdot (\mathtt{o} \to \mathtt{o} \to \mathtt{o})$ and $\mathtt{o} \to \mathtt{o} \to \mathtt{o}$ respectively. Thus, the remaining sequence "$\lambda_{\mathtt{o}\to\mathtt{o}}\, 1\,0\,0\,0$" consists of two substrings, which represent M_0 and M_1 respectively. Since the head is $\lambda_{\mathtt{o}\to\mathtt{o}}$, we know M_0 must be of the form $(\lambda.M_{0,0})M_{0,1} \cdots M_{0,k}$. Using the type information again, we can find that $k = 0$, hence M_0 is actually of the form $\lambda : \mathtt{o}.M_{0,0}$, where $M_{0,0}$ has type $\mathtt{o}$ under $\mathtt{o} \cdot (\mathtt{o} \to \mathtt{o} \to \mathtt{o}) \cdot (\mathtt{o} \to \mathtt{o} \to \mathtt{o})$. Since the head of the remaining sequence is 1, $M_{0,0}$ must be of the form $1\, M_{0,0,0}\, M_{0,0,1}$ where both $M_{0,0,0}$ and $M_{0,0,1}$ have type $\mathtt{o}$. By continuing this reasoning, we can recover the original term $(\lambda : \mathtt{o} \to \mathtt{o} \to \mathtt{o}.\lambda : \mathtt{o}.1\,0\,0)0$.

REMARK 3.1. *A reader may wonder why we do not annotate each λ with the type of the variable bound by the λ-abstraction, instead of the type of the entire abstraction. Decoding would not work if we did so. Consider the term $\lambda : \mathtt{o}.(2\,0)\,0$, which has type $\mathtt{o} \to \mathtt{o}$ under the environment $\mathtt{o} \cdot (\mathtt{o} \to \mathtt{o} \to \mathtt{o})$. If λ were annotated with the type of the bound variable, it would be encoded to:*

$$\lambda_{\mathtt{o}}\, 2\,0\,0.$$

Given the type $\mathtt{o} \to \mathtt{o}$ of the term and the type environment $\mathtt{o} \cdot (\mathtt{o} \to \mathtt{o} \to \mathtt{o})$, we do not know whether the sequence should be decoded to: $\lambda : \mathtt{o}.((2\,0)\,0)$, or $(\lambda : \mathtt{o}.2\,0)0$. In fact, both terms have type $\mathtt{o} \to \mathtt{o}$ under the environment $\mathtt{o} \cdot (\mathtt{o} \to \mathtt{o} \to \mathtt{o})$.

We now define the encoding/decoding functions formally. Let Λ be the set of simply-typed λ-terms, $\mathbf{Ty}$ be the set of types, and $\mathbf{Sym}$ be the set of symbols $\mathbf{Nat} \cup \{\lambda_\tau \mid \tau \in \mathbf{Ty}\}$. We also write $\mathbf{TEnv}$ for the set of type environments, i.e., $\mathbf{Ty}^*$. We write $\mathit{TypeOf}(\Gamma, M)$ for the type τ such that $\Gamma \vdash M : \tau$; note that such τ is unique if it exists.

The encoding and decoding functions EncS and DecS are shown in Figure 1. The encoding function EncS is trivial; it just traverses the term from left to right, and lists up the encountered symbols. The decoding function DecS takes a sequence of symbols s along with the type environment Γ and the type τ of the original term as input. It returns a pair (M, s') such that $\Gamma \vdash M : \tau$ and $s = \mathit{EncS}(\Gamma, M) \cdot s'$. It may fail when the input sequence is invalid. For example, $\mathit{DecS}(0\,1\,1\,0\,1\,1, (\mathtt{o} \to \mathtt{o} \to \mathtt{o}) \cdot \mathtt{o}, \mathtt{o})$ should return $((0\,1)\,1, 0\,1\,1)$. When the head of an input sequence is an index n, then $\Gamma(n)$ should be of the form $\tau_1 \to \cdots \to \tau_k \to \tau$ for some k, and the original term must be of the form $n\, M_1 \cdots M_k$, where each M_i has type τ_i. Thus, the first k segments of the remaining sequence s_0 must represent $M_1, \ldots, M_k$. The sequence does not contain any separator of the segments, but by reading s_0 from the head, we can correctly recognize $M_1, \ldots, M_k$. The case for λ works in a similar manner.

The following theorem guarantees the correctness of EncS and DecS.

THEOREM 1. *Suppose $\Gamma \vdash M : \tau$.*
Then, $\mathit{DecS}(\mathit{EncS}(\Gamma, M), \Gamma, \tau) = (M, \epsilon)$.

$\mathit{EncS} \in \mathbf{TEnv} \times \Lambda \rightharpoonup \mathbf{Sym}^*$:
$\mathit{EncS}(\Gamma, n) = n$
$\mathit{EncS}(\Gamma, \lambda : \tau_0.M) = \lambda_{\mathit{TypeOf}(\Gamma, \lambda:\tau_0.M)} \cdot \mathit{EncS}(\tau_0 \cdot \Gamma, M)$
$\mathit{EncS}(\Gamma, M_1 M_2) = \mathit{EncS}(\Gamma, M_1) \cdot \mathit{EncS}(\Gamma, M_2)$

$\mathit{DecS} \in \mathbf{Sym}^* \times \mathbf{TEnv} \times \mathbf{Ty} \rightharpoonup \Lambda \times \mathbf{Sym}^*$:
$\mathit{DecS}(\epsilon, \Gamma, \tau) = \mathbf{fail}$
$\mathit{DecS}(n \cdot s_0, \Gamma, \tau) =$
  `if` $\Gamma(n)$ is of the form $\tau_1 \to \cdots \to \tau_k \to \tau$ $(k \geq 0)$
  `then let` $(M_1, s_1) = \mathit{DecS}(s_0, \Gamma, \tau_1)$ `in`
    $\cdots$
      `let` $(M_k, s_k) = \mathit{DecS}(s_{k-1}, \Gamma, \tau_k)$ `in`
        $(nM_1 \cdots M_k, s_k)$
  `else` **fail**
$\mathit{DecS}(\lambda_{\tau' \to \tau''} \cdot s, \Gamma, \tau) =$
  `if` $\tau' \to \tau''$ is of the form $\tau_1 \to \cdots \to \tau_k \to \tau$ $(k \geq 0)$
  `then let` $(M_0, s_0) = \mathit{DecS}(s, \tau' \cdot \Gamma, \tau'')$ `in`
      `let` $(M_1, s_1) = \mathit{DecS}(s_0, \Gamma, \tau_1)$ `in`
        $\cdots$
      `let` $(M_k, s_k) = \mathit{DecS}(s_{k-1}, \Gamma, \tau_k)$ `in`
        $((\lambda : \tau_1.M_0)M_1 \cdots M_k, s_k)$
  `else` **fail**

Figure 1. Encoding/decoding functions for tagless coding

Proof. We show that $\mathit{DecS}(\mathit{EncS}(\Gamma, M) \cdot s, \Gamma, \tau) = (M, s)$ holds for every $s \in \mathbf{Sym}^*$ by induction on the structure of M (according to the inductive definition of λ-terms in Remark 2.1). The theorem follows as a special case where $s = \epsilon$.

- Case $M = nM_1 \cdots M_k$ (where k may be 0): In this case, we have:

  $$\begin{array}{l}\Gamma(n) = \tau_1 \to \cdots \to \tau_k \to \tau \\ \Gamma \vdash M_i : \tau_i \text{ for each } i \in \{1, \ldots, k\} \\ \mathit{EncS}(\Gamma, M) = n\mathit{EncS}(\Gamma, M_1) \cdots \mathit{EncS}(\Gamma, M_k).\end{array}$$

  By the induction hypothesis, we have:

  $$\begin{array}{l}\mathit{DecS}(\mathit{EncS}(\Gamma, M_1) \cdots \mathit{EncS}(\Gamma, M_k)s, \Gamma, \tau_1) = \\ \quad (M_1, \mathit{EncS}(\Gamma, M_2) \cdots \mathit{EncS}(\Gamma, M_k)s) \\ \cdots \\ \mathit{DecS}(\mathit{EncS}(\Gamma, M_k)s, \Gamma, \tau_k) = (M_k, s).\end{array}$$

  Thus, by the case of $n \cdot s_0$ in the definition of DecS, $\mathit{DecS}(\mathit{EncS}(\Gamma, M) \cdot s, \Gamma, \tau)$ returns $(n\, M_1 \cdots M_k, s)$, i.e., (M, s).

- Case $M = (\lambda : \tau'.M_0)M_1 \cdots M_k$ (where k may be 0): In this case, we have:

  $$\begin{array}{l}\Gamma \vdash \lambda : \tau'.M_0 : \tau_1 \to \cdots \to \tau_k \to \tau \\ \Gamma \vdash M_i : \tau_i \text{ for each } i \in \{1, \ldots, k\} \\ \mathit{EncS}(\Gamma, M) = \\ \quad \lambda_{\tau_1 \to \cdots \to \tau_k \to \tau} \cdot \mathit{EncS}(\tau' \cdot \Gamma, M_0) \cdot \\ \qquad \mathit{EncS}(\Gamma, M_1) \cdots \mathit{EncS}(\Gamma, M_k)\end{array}$$

  By the first condition, we have $\tau' \cdot \Gamma \vdash M_0 : \tau''$, where $\tau' \to \tau'' = \tau_1 \to \cdots \to \tau_k \to \tau$. By the induction hypothesis, we have:

  $$\begin{array}{l}\mathit{DecS}(\mathit{EncS}(\tau' \cdot \Gamma, M_0)\mathit{EncS}(\Gamma, M_1) \cdots \mathit{EncS}(\Gamma, M_k)s, \\ \quad \tau' \cdot \Gamma, \tau'') = \\ \qquad (M_0, \mathit{EncS}(\Gamma, M_1) \cdots \mathit{EncS}(\Gamma, M_k)s) \\ \mathit{DecS}(\mathit{EncS}(\Gamma, M_1) \cdots \mathit{EncS}(\Gamma, M_k)s, \Gamma, \tau_1) = \\ \qquad (M_1, \mathit{EncS}(\Gamma, M_2) \cdots \mathit{EncS}(\Gamma, M_k)s) \\ \cdots \\ \mathit{DecS}(\mathit{EncS}(\Gamma, M_k)s, \Gamma, \tau_k) = (M_k, s)\end{array}$$

Thus, by the case of $\lambda_{\tau'\to\tau''} \cdot s_0$ in the definition of $DecS$, $DecS(EncS(\Gamma, M) \cdot s, \Gamma, \tau)$ returns

$$((\lambda : \tau'.M_0)M_1 \cdots M_k, s),$$

i.e., (M, s) as required. □

3.2 Bit Encoding

By the encoding in the previous subsection, a λ-term has been converted to a triple (s, Γ, τ) where s is a sequence of symbols, Γ is a type environment, and τ is a type. The triple is further converted to a bit string consisting of (the bit representations of) the following parts.

(i) A table of types, which associates each type occurring in s with a unique identifier (which is just a number).

(ii) Γ and τ.

(iii) the sequence s of symbols.

Additionally, when the names of free variables are important, we prepare a symbol table that maps (de Bruijn index of) each free variable to its name. The symbol table can be encoded by using an off-the-shelf string compressor; in the experiments reported later, we used bzip2 [11].

For part (i), each type τ is represented as the bit sequence $[\![\tau]\!]$, defined by:

$$[\![\mathsf{o}]\!] = 0 \quad [\![\, \tau_1 \to \tau_2]\!] = 1\, [\![\, \tau_1 \,]\!]\, [\![\tau_2 \,]\!]\,.$$

For example, $(\mathsf{o} \to \mathsf{o}) \to \mathsf{o}$ and $\mathsf{o} \to (\mathsf{o} \to \mathsf{o})$ are represented as 11000 and 10100 respectively. The type table is just a sequence $[\![\tau_0]\!] \cdots [\![\tau_{m-1}]\!]$, where $\{\tau_0, \ldots, \tau_{m-1}\}$ is the set of types occurring in s, $\{\tau' \mid \lambda_{\tau'} \text{ occurs in } s\}$. Note that no separator is required since $[\![\tau]\!]$ is a prefix code. The sequence assigns the identifier i to each τ_i.

Part (ii) is just a bit sequence $[\![\tau \,]\!]\, [\![\tau_1 \,]\!] \cdots [\![\, \tau_n]\!]$, where $\Gamma = \tau_1 \cdots \tau_n$. We here assume that parts (i) and (ii) above are small compared with the length of s, which is typically the case when the λ-term is large, but the set of types occurring in the term is small; otherwise we may further compress the bit strings for part (i) and (ii) with a string compressor.

For Part (iii), we first replace each λ_τ with the identifier of τ prepared in (i) above, and shift each variable index n in s by m, where $m-1$ is the largest type identifier. Thus, we have a sequence of natural numbers, which is encoded into a bit string by using an (adaptive) range coding [8].

EXAMPLE 1. *Recall the λ-term $(\lambda : \mathsf{o} \to \mathsf{o} \to \mathsf{o}.\lambda : \mathsf{o}.(1\,0)\,0)0$, which has been converted to*

$$\lambda_{(\mathsf{o}\to\mathsf{o}\to\mathsf{o})\to\mathsf{o}\to\mathsf{o}}\, \lambda_{\mathsf{o}\to\mathsf{o}}\, 1\,0\,0\,0,$$

in Section 3.1. The part (i) is represented by:

$$\underbrace{110100100}_{[\![(\mathsf{o}\to\mathsf{o}\to\mathsf{o})\to\mathsf{o}\to\mathsf{o}]\!]}\ \underbrace{100}_{[\![\mathsf{o}\to\mathsf{o}]\!]}.$$

The part (ii) is represented by:

$$\underbrace{100}_{[\![\mathsf{o}\to\mathsf{o}]\!]}\ \underbrace{10100}_{[\![\mathsf{o}\to\mathsf{o}\to\mathsf{o}]\!]}.$$

For the part (iii), the sequence of symbols is first converted to the following sequence of numbers:

$$0\,1\,3\,2\,2\,2$$

by replacing $\lambda_{(\mathsf{o}\to\mathsf{o}\to\mathsf{o})\to\mathsf{o}\to\mathsf{o}}$ and $\lambda_{\mathsf{o}\to\mathsf{o}}$ with their identifiers 0 and 1, and shifting each de Bruijn index by 2. We then apply adaptive range coding to the sequence of numbers.

REMARK 3.2. *Actually, before applying range coding for part (iii), we can also apply the following optimization. For each position of the sequence s, we know the type of the term encoded by the sequence starting from the position, and use it to exclude some symbols out of consideration and renumber the symbol at the position: if the set of numbers that may occur at the position is $\{a_0, \ldots, a_{k-1}\}$ with $a_0 < \cdots < a_{k-1}$, and the actual number at the position is a_j, then we can renumber a_j to j. As a concrete example, consider the term $(1\,2)\,0$, which has type o under the environment $\mathsf{o} \cdot ((\mathsf{o} \to \mathsf{o}) \to \mathsf{o} \to \mathsf{o}) \cdot (\mathsf{o} \to \mathsf{o})$. By the first phase, we get a sequence $1\,2\,0$. At the first position, any of $1, 2, 0$ may occur since the sequence encodes a term of type o. At the second position, however, the substring $2 \cdots$ encodes a term of type $\mathsf{o} \to \mathsf{o}$; hence, we know that 0 may not occur. Using that information, we can renumber the index by excluding 0, obtaining $1\,1\,0$ instead of $1\,2\,0$. When decoding the sequence, we can find, from type information, that 0 has been excluded out at the second position by the renumbering; thus we can recover the original sequence $1\,2\,0$, and apply the decoding function in the previous section to get the term $(1\,2)\,0$. In this manner, we can encode a sequence s with a smaller set of numbers, which allows range coding to produce a more compact code.*

4. Second Approach: Grammar-based Coding

In the approach presented in Section 3, a λ-term has been first converted to a sequence of numbers, and then an off-the-shelf string compressor has been used to convert the sequence to a bit string. By using a range coder [8] as the string compressor, we can take into account the probability distribution of occurrences of the numbers in the sequence. As discussed in Remark 3.2, however, the numbers that may occur at each position of the sequence differ, depending on the type of the subterm encoded by the substring starting at the position. Remark 3.2 suggested some optimization to take such contextual information into account, but as long as we use a string compressor as a backend, it is difficult to fully exploit the information. The backend string compressor recognizes the sequence of numbers just as an element of $\{0, \ldots, m\}^*$ where m is the largest number, and assigns a code to each of them, but some of the sequences actually never occur and the bits for representing them are wasted.

To overcome the problem above, in the second approach, we use, as the backend compressor, a compressor that is parametrized by a context-free grammar, and assigns bit codes to only sequences that may be generated by the grammar. By appropriately designing a grammar, we can assign bit codes to only sequences that represent simply-typed λ-terms. Using the grammar also makes it easy to impose various restrictions on the syntax of simply-typed λ-terms being considered, enabling even more compact coding.

As the backend compressor that takes a grammar into account, we prepared one based on Cameron [2]'s encoding scheme, which is a variation of arithmetic coding. In the rest of this section, we first review Cameron's scheme [2] in Section 4.1. We then describe how to utilize it for the bit encoding of simply-typed λ-terms in Section 4.2. We discuss further optimizations in Section 4.3.

4.1 Cameron's Encoding Scheme for CFG

We explain the idea of Cameron's encoding scheme by using an example. Suppose that we wish to convert an arithmetic expression, generated by the following grammar, into a bit code.

$$E \to E + F \mid F$$
$$F \to F \times a \mid a$$

If we view an arithmetic expression simply as a string consisting of $a, +, \times$, and apply a string compressor, then useless bit codes are allocated to non-arithmetic expressions such as "$aa\times$" and

"$+a+a$". That can be avoided by encoding, instead of an arithmetic expression, a parse tree or a derivation sequence that derives the expression. For example, for the expression $a + a \times a$, we may think of the following sequence:

$$[E \to E + F]\,[E \to F]\,[F \to a]\,[F \to F \times a]\,[F \to a]$$

which describes what production rules have been used in the leftmost derivation to generate $a + a \times a$. Let us assign 0 to the left production rule of each non-terminal, and 1 to the right one. Then, the above sequence can be encoded as: 01101. Note that the same bit can be used for the left rules of E and F, since at each position, we know which non-terminal is expanded (by the assumption that the sequence represents the leftmost derivation).

For PL researchers, it may be easier to understand that we are considering data of the following algebraic data type (written in the syntax of OCaml), instead of a string over $\{a, +, \times\}$:

```
type exp = Plus of exp * fact | Factor of fact
and fact = Times of fact * var | Var of var
and var = A.
```

The expression $a + a \times a$ is expressed by:

$$\texttt{Plus}(\texttt{Factor}(\texttt{Var}(\texttt{A})), \texttt{Times}(\texttt{Var}(\texttt{A}), \texttt{A})).$$

It is then converted to the bit code 01101, assigning 0 to `Plus` and `Times` and 1 to `Factor` and `Var`. In this manner, we can avoid allocating bit codes to non-expressions.

A little more generally, suppose that a grammar $\mathcal{G}$ consists of the set of production rules:

$$\{A_i \to X_{i,j,1} \cdots X_{i,j,k_{i,j}} \mid i \in \{1, \ldots, m\}, j \in \{1, \ldots, n_i\}\}$$

where each $X_{i,j,k}$ is a terminal or a non-terminal symbol, and A_1 is the start symbol. The functions $Enc_{\mathcal{G}}$ and $Dec_{\mathcal{G}}$ for encoding and decoding a derivation sequence are given by:

$$\begin{array}{l}
Enc_{\mathcal{G}}([A_i \to X_{i,j,1} \cdots X_{i,j,k_{i,j}}]\pi) = j \cdot Enc_{\mathcal{G}}(\pi) \\
Dec_{\mathcal{G}}(s) = DecAux_{\mathcal{G}}(A_1, s) \\
DecAux_{\mathcal{G}}(A_i W, js) = \\
\quad [A_i \to X_{i,j,1} \cdots X_{i,j,k_{i,j}}] DecAux_{\mathcal{G}}(X_{i,j,1} \cdots X_{i,j,k_{i,j}} W, s) \\
DecAux_{\mathcal{G}}(aW, s) = DecAux_{\mathcal{G}}(W, s) \\
DecAux_{\mathcal{G}}(\epsilon, \epsilon) = \epsilon.
\end{array}$$

It is obvious that for any valid leftmost derivation sequence π of grammar $\mathcal{G}$, $Dec_{\mathcal{G}}(Enc_{\mathcal{G}}(\pi)) = \pi$.

In the explanation above, we have used an integer to represent each choice of a production rule. Cameron's scheme is actually based on arithmetic coding [14], and can assign a fractional bit for each production rule, reflecting the frequency of occurrences of each production rule. For example, suppose that the probabilities that the rules $F \to F \times a$ and $F \to a$ are used in rewriting F are $1/3$ and $2/3$ respectively (as in the case $a + a \times a$ above). Then, for a sufficiently large input, the bit lengths assigned to $F \to F \times a$ and $F \to a$ are $-\log(1/3) \approx 1.585$ and $-\log(2/3) \approx 0.585$ respectively. Appendix A describes more detail on Cameron's scheme.

We assume below that we are given Cameron's encoder and decoder $EncC_{\mathcal{G}}$ and $DecC_{\mathcal{G}}$ such that $DecC_{\mathcal{G}}(EncC_{\mathcal{G}}(w)) = w$ for every $w \in \mathcal{L}(\mathcal{G})$, where $\mathcal{G}$ is a context-free grammar and $\mathcal{L}(\mathcal{G})$ denotes the language generated by $\mathcal{G}$. Thus, we just need to develop an appropriate context-free grammar $\mathcal{G}$ for simply-typed λ-terms, which is discussed in the next subsection.

4.2 Grammar-based Encoding of Simply-Typed λ-Terms

As mentioned above, in order to apply Cameron's bit encoding scheme to simply-typed λ-terms, it suffices to prepare a context-free grammar to describe well-typed terms. Fortunately, typing rules of the simply-typed λ-calculus may be read as production rules of CFG. For example, the rule for applications:

$$\frac{\Gamma \vdash M : \tau' \to \tau \qquad \Gamma \vdash N : \tau'}{\Gamma \vdash MN : \tau}$$

may be read as the production rule:

$$M_\tau \to M_{\tau' \to \tau} M_{\tau'}$$

for each τ, τ', where M_τ is a non-terminal for generating terms of type τ. In this way, the typing rules may be interpreted as an "infinite" context-free grammar consisting of an infinite set of production rules. For a given term, we may consider fixed, finite sets of types and variables; thus, we can prepare an ordinary context-free grammar consisting of only a finite number of production rules.

To reduce the number of production rules for variables, we slightly modify de Bruijn notation; each variable is annotated with its type, and numbered separately for each type. Thus, for example, the term $\lambda x : \mathtt{o} \to \mathtt{o}.\lambda y : \mathtt{o} \to \mathtt{o}.\lambda z : \mathtt{o}.x(y\,z)$ is represented by $\lambda{:}\mathtt{o} \to \mathtt{o}.\lambda{:}\mathtt{o} \to \mathtt{o}.\lambda{:}\mathtt{o}.1_{\mathtt{o}\to\mathtt{o}}(0_{\mathtt{o}\to\mathtt{o}}\,0_{\mathtt{o}})$, instead of $\lambda{:}\mathtt{o} \to \mathtt{o}.\lambda{:}\mathtt{o} \to \mathtt{o}.\lambda : \mathtt{o}.2_{\mathtt{o}\to\mathtt{o}}(1_{\mathtt{o}\to\mathtt{o}}\,0_{\mathtt{o}})$. The index $0_{\mathtt{o}\to\mathtt{o}}$ in the new representation refers to the inner-most binder for a variable of type $\mathtt{o} \to \mathtt{o}$, which is the second λ.

The production rules for simply-typed λ-terms are given by:

$$\begin{array}{rcl}
M_\tau & \to & n_{\tau_1 \to \cdots \to \tau_k \to \tau} M_{\tau_1} \cdots M_{\tau_k} \\
& \mid & (\lambda : \tau'.M_{\tau''}) M_{\tau_1} \cdots M_{\tau_k}
\end{array}$$

These are actually "templates", from which the actual production rules can be obtained by appropriately instantiating $\tau, \tau_1, \ldots, \tau_k, \tau', \tau''$ to concrete types. In the second line, $\tau' \to \tau'' = \tau_1 \to \cdots \to \tau_k \to \tau$ should hold in each instantiation of the template.

In the case of the term $\lambda{:}\mathtt{o} \to \mathtt{o}.\lambda{:}\mathtt{o} \to \mathtt{o}.\lambda{:}\mathtt{o}.1_{\mathtt{o}\to\mathtt{o}}(0_{\mathtt{o}\to\mathtt{o}}\,0_{\mathtt{o}})$, the actual grammar $\mathcal{G}_1$ consists of the following production rules:

$$\begin{array}{l}
M_{(\mathtt{o}\to\mathtt{o})\to(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}} \to \lambda : \mathtt{o} \to \mathtt{o}.M_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}} \\
M_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}} \to \lambda : \mathtt{o} \to \mathtt{o}.M_{\mathtt{o}\to\mathtt{o}} \\
\qquad\qquad \mid\ (\lambda : \mathtt{o} \to \mathtt{o}.M_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}}) M_{\mathtt{o}\to\mathtt{o}} \\
M_{\mathtt{o}\to\mathtt{o}} \to 0_{\mathtt{o}\to\mathtt{o}} \mid 1_{\mathtt{o}\to\mathtt{o}} \mid \lambda : \mathtt{o}.M_{\mathtt{o}} \\
\qquad \mid\ (\lambda : \mathtt{o} \to \mathtt{o}.M_{\mathtt{o}\to\mathtt{o}}) M_{\mathtt{o}\to\mathtt{o}} \\
\qquad \mid\ (\lambda : \mathtt{o} \to \mathtt{o}.M_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}}) M_{\mathtt{o}\to\mathtt{o}} M_{\mathtt{o}\to\mathtt{o}} \\
M_{\mathtt{o}} \to 0_{\mathtt{o}} \mid 0_{\mathtt{o}\to\mathtt{o}} M_{\mathtt{o}} \mid 1_{\mathtt{o}\to\mathtt{o}} M_{\mathtt{o}}
\end{array}$$

The rightmost derivation of the term is represented by:

$$\begin{array}{l}
[M_{(\mathtt{o}\to\mathtt{o})\to(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}} \to \lambda : \mathtt{o} \to \mathtt{o}.M_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}}] \\
[M_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}} \to \lambda : \mathtt{o} \to \mathtt{o}.M_{\mathtt{o}\to\mathtt{o}}] \\
[M_{\mathtt{o}\to\mathtt{o}} \to \lambda : \mathtt{o}.M_{\mathtt{o}}] \\
[M_{\mathtt{o}} \to 1_{\mathtt{o}\to\mathtt{o}} M_{\mathtt{o}}] \\
[M_{\mathtt{o}} \to 0_{\mathtt{o}\to\mathtt{o}} M_{\mathtt{o}}] \\
[M_{\mathtt{o}} \to 0_{\mathtt{o}}].
\end{array}$$

If we abbreviate the production rule

$$M_\tau \to n_{\tau_1 \to \cdots \to \tau_k \to \tau} M_{\tau_1} \cdots M_{\tau_k}$$

to $[n_{\tau_1 \to \cdots \to \tau_k \to \tau}]$ and

$$M_\tau \to (\lambda : \tau'.M_{\tau''}) M_{\tau_1} \cdots M_{\tau_k}$$

to $[\lambda_{\tau' \to \tau''}]$, then the above sequence becomes:

$$[\lambda_{(\mathtt{o}\to\mathtt{o})\to(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}}][\lambda_{(\mathtt{o}\to\mathtt{o})\to\mathtt{o}\to\mathtt{o}}][\lambda_{\mathtt{o}\to\mathtt{o}}][1_{\mathtt{o}\to\mathtt{o}}][0_{\mathtt{o}\to\mathtt{o}}][0_{\mathtt{o}}].$$

This corresponds to the tagless representation in Section 3; the differences are: (i) we use arithmetic coding to take into account the frequency of occurrences of each production rule whereas in the previous section, we have just passed the sequence to a string compressor, ignoring the nature of each symbol in the sequence, and (ii) we can further optimize the grammar, as discussed in the next subsection.

REMARK 4.1. *The context-free grammar described above may generate invalid terms. For example, the grammar $\mathcal{G}_1$ obtained for*

$\lambda:\mathsf{o}\to\mathsf{o}.\lambda:\mathsf{o}\to\mathsf{o}.\lambda:\mathsf{o}.1_{\mathsf{o}\to\mathsf{o}}(0_{\mathsf{o}\to\mathsf{o}}\,0_{\mathsf{o}})$ *also generates a term like* $\lambda:\mathsf{o}\to\mathsf{o}.0_{\mathsf{o}}$*, which is not a closed term. This is because our grammar ignores type environments. To address this problem, we modify Cameron's encoding scheme to take into account which production rule may be used at each position, and avoid allocating bits for unused production rules. This is achieved by adjusting the probability distribution of production rules. For example, for* $\mathcal{G}_1$*, suppose that the probability distribution of the production rules for* M_{o} *is given by:*

$$\begin{array}{ll} M_{\mathsf{o}}\to 0_{\mathsf{o}} & 1/2\\ M_{\mathsf{o}}\to 0_{\mathsf{o}\to\mathsf{o}}M_{\mathsf{o}} & 1/4\\ M_{\mathsf{o}}\to 1_{\mathsf{o}\to\mathsf{o}}M_{\mathsf{o}} & 1/4\end{array}$$

In a context where $1_{\mathsf{o}\to\mathsf{o}}$ *may not occur, we encode production rules based on the following redistributed probabilities.*

$$\begin{array}{ll} M_{\mathsf{o}}\to 0_{\mathsf{o}} & 2/3\\ M_{\mathsf{o}}\to 0_{\mathsf{o}\to\mathsf{o}}M_{\mathsf{o}} & 1/3\\ M_{\mathsf{o}}\to 1_{\mathsf{o}\to\mathsf{o}}M_{\mathsf{o}} & 0\end{array}$$

In this manner, we can avoid allocating bit codes for invalid λ*-terms.*

An alternative approach would have been to take type environments into account explicitly in a grammar. We could take $M_{\Gamma,\tau}$ *as a non-terminal for generating "terms of type* τ *under* Γ*", and prepare production rules like*

$$M_{\Gamma,\tau}\to M_{\Gamma,\tau'\to\tau}M_{\Gamma,\tau'}$$

and

$$M_{\tau_1\to\tau_2}\to\lambda:\tau_1.M_{\tau_1\cdot\Gamma,\tau_2}.$$

We did not do so since the number of production rules would become too large to be effective with Cameron's encoding scheme; the overhead of representing the grammar (cf. Section 4.4) would outweigh the benefit of Cameron's scheme.

4.3 Optimizations Based on $\beta\eta$-Equivalence

We have so far considered bit encoding of λ-terms up to α-equivalence. In the context of higher-order compression [6], we need not necessarily distinguish between $\beta\eta$-equivalent terms, as a λ-term is used to describe the tree data obtained by reducing the λ-term to its $\beta\eta$-normal form. This brings a further opportunity of optimizations; for example, we may allocate the same bit code for $(\lambda x.L)MN$ and $(\lambda x.(LN))M$(assuming that x does not occur in N). Our grammar-based encoding is convenient for applying such optimizations.

As an extreme approach, it would be possible to consider β-normal forms and allocate bit codes only for them. This, however, results in allocating a large code for a term having a large normal form, which is against the goal of higher-order compression to use λ-terms as compressed representations of tree data. Thus, we may allocate to M the code for N only if M is $\beta\eta$-equivalent to N, *and* the size of N is not larger than that of M. To this end, we prepare a few transformation rules that preserve the $\beta\eta$-equivalence, and do not increase the term size, and construct a grammar that over-approximates the set of normal forms with respect to the transformation rules. We can then convert a given term to a bit code by first reducing the term to a normal form (with respect to the transformation rules), and allocate a code based on the grammar.

We first prepare the following transformation rule:[2]

$$(\lambda x.L)MN\longrightarrow(\lambda x.LN)M\qquad(x\text{ is not free in }N)$$

(which may be easier to understand if it is written $(\mathbf{let}\ x=M\ \mathbf{in}\ L)N\longrightarrow\mathbf{let}\ x=M\ \mathbf{in}\ (LN)$, where $\mathbf{let}\ x=M\ \mathbf{in}\ L$ is an abbreviation of $(\lambda x.L)M$). Clearly, the transformation preserves the $\beta\eta$-equivalence. Based on the rule, we can optimize the grammar in Section 4.2 to:

$$\begin{array}{rcl} M_\tau & \to & n_{\tau_1\to\cdots\to\tau_k\to\tau}M_{\tau_1}\cdots M_{\tau_k}\\ & \mid & (\lambda:\tau'.M_\tau)M_{\tau'}\\ & \mid & \lambda:\tau'.M_{\tau''}\ (\text{where }\tau=\tau'\to\tau'').\end{array}$$

Here, we have excluded out terms of the form $(\lambda x.L)MN$, In this manner, we can avoid allocating redundant codes to those terms.

We also consider the transformation rules:

$$(\lambda x.M)y\longrightarrow[y/x]M$$

and

$$(\lambda x.L)((\lambda y.M)N)\longrightarrow(\lambda y.(\lambda x.L)M)N\qquad(y\text{ does not occur in }L).$$

The first rule is a restricted form of the β-reduction rule, where the argument is restricted to a variable, so that the reduction does not increase the term size. The latter may be easier to understand when presented using let-expressions:

$$\mathbf{let}\ x=(\mathbf{let}\ y=N\ \mathbf{in}\ M)\ \mathbf{in}\ L\longrightarrow\mathbf{let}\ y=N\ \mathbf{in}\ \mathbf{let}\ x=M\ \mathbf{in}\ L.$$

Based on the transformation rules above, we can further optimize the grammar to:

$$\begin{array}{rcl} M_\tau & \to & n_{\tau_1\to\cdots\to\tau_k\to\tau}M_{\tau_1}\cdots M_{\tau_k}\\ & \mid & (\lambda:\tau'.M_\tau)N_{\tau'}\\ & \mid & \lambda:\tau'.M_{\tau''}\ (\text{where }\tau=\tau'\to\tau'')\\ N_\tau & \to & n_{\tau_1\to\cdots\to\tau_k\to\tau}M_{\tau_1}\cdots M_{\tau_k}\ (\text{where }k>0)\\ & \mid & \lambda:\tau'.M_{\tau''}\ (\text{where }\tau=\tau'\to\tau'').\end{array}$$

Here, we have excluded out terms of the form $(\lambda x.M)y$ or $(\lambda x.L)((\lambda y.M)N)$.

We also wish to take into account the equivalence:

$$\begin{array}{l}(\mathbf{let}\ x=L\ \mathbf{in}\ \mathbf{let}\ y=M\ \mathbf{in}\ N)\\ \quad=_\beta(\mathbf{let}\ y=M\ \mathbf{in}\ \mathbf{let}\ x=L\ \mathbf{in}\ N),\end{array}$$

assuming x and y do not occur in L, M, and allocate the same bit code for both terms. Without such optimization, $n!$ different bit codes would be allocated to terms of the form:

$$\mathbf{let}\ x_{\theta1}=M_{\theta1}\ \mathbf{in}\ \cdots\mathbf{let}\ x_{\theta n}=M_{\theta n}\ \mathbf{in}\ N,$$

where x_i does not occur in none of M_j's, and θ is a permutation on $\{1,\ldots,n\}$, although the terms are β-equivalent to each other. Applying such an optimization by rewriting a grammar is difficult, however, since both sides of the equivalence

$$\begin{array}{l}(\mathbf{let}\ x=L\ \mathbf{in}\ \mathbf{let}\ y=M\ \mathbf{in}\ N)\\ \quad=_\beta(\mathbf{let}\ y=M\ \mathbf{in}\ \mathbf{let}\ x=L\ \mathbf{in}\ N),\end{array}$$

have the same syntactic form.

To address the issue above, we consider a certain total order $<$ on λ-terms, and reorder let-expressions, so that $L<M$ always holds in $\mathbf{let}\ x=L\ \mathbf{in}\ \mathbf{let}\ y=M\ \mathbf{in}\ N$. Then, we take that constraint into account during range coding, by adjusting probability distributions as in Remark 4.1. For example, assuming the order $0_{\mathsf{o}\to\mathsf{o}}M<1_{\mathsf{o}\to\mathsf{o}}M'$, when encoding a term of the form $\mathbf{let}\ x=1_{\mathsf{o}\to\mathsf{o}}M'\ \mathbf{in}\ \mathbf{let}\ y=K\ \mathbf{in}\ L$, we set the probability that K is of the form $0_{\mathsf{o}\to\mathsf{o}}M$ to 0.

4.4 Bit Representation of Grammar Information

The bit representation of a λ-term consists of the bit sequence obtained by the grammar-based arithmetic encoding described in Sections 4.2 and 4.3, and information about the grammar used for the encoding. Since we have prepared an adaptive, grammar-

[2] For the readability, we use the standard notation for λ-terms instead of de Bruijn notation.

based range coder customized for λ-terms, we can assume a fixed grammar template, such as the one given in Section 4.3:

$$\begin{array}{rcl} M_\tau & \to & n_{\tau_1 \to \cdots \to \tau_k \to \tau} M_{\tau_1} \cdots M_{\tau_k} \\ & | & (\lambda : \tau'.M_\tau) M_{\tau'} \\ & | & \lambda : \tau'.M_{\tau''} \text{ (where } \tau = \tau' \to \tau'') \end{array}$$

Since concrete production rules are obtained by instantiating types and indices n, we still need to specify which instantiations actually exist. To this end, we prepare the following data:

- A sequence T_N of types, for which a production rule for M_τ exists.
- A sequence T_V of types, for which a variable n_τ is used for some n.
- A map $V_{\max}$, which describes, for each τ in T_N and τ' in T_V, the maximum index n such that a rule of the form
$$M_\tau \to n_{\tau'} M_{\tau_1} \cdots M_{\tau_k}$$
exists.
- A bit map V, which describes, for each τ in T_N, τ' in T_V, and $m \leq V_{\max(\tau,\tau')}$, whether a rule of the form
$$M_\tau \to n_{\tau'} M_{\tau_1} \cdots M_{\tau_k}$$
exists.
- A bit map L, which describes, for each τ in T_N and τ' in T_V, whether the rule
$$(\lambda : \tau'.M_\tau) M_{\tau'}$$
exists.
- A bit map A, which describes, for each $\tau' \to \tau''$ in T_N, whether the rule
$$M_{\tau' \to \tau''} \to \lambda : \tau'.M_{\tau''}$$
exists.

Note that, since we use adaptive range coding, we need not prepare information on how often each production rule is used. We represent all the data above as a bit sequence, and then compress it by using an off-the-shelf string compressor (bzip2 [11], in the experiments reported in the next section).

The more optimized version of the grammar:

$$\begin{array}{rcl} M_\tau & \to & n_{\tau_1 \to \cdots \to \tau_k \to \tau} M_{\tau_1} \cdots M_{\tau_k} \\ & | & (\lambda : \tau'.M_\tau) N_{\tau'} \\ & | & \lambda : \tau'.M_{\tau''} \text{ (where } \tau = \tau' \to \tau'') \\ N_\tau & \to & n_{\tau_1 \to \cdots \to \tau_k \to \tau} M_{\tau_1} \cdots M_{\tau_k} \text{ (where } k > 0) \\ & | & \lambda : \tau'.M_{\tau''} \text{ (where } \tau = \tau' \to \tau''), \end{array}$$

can be represented in a similar manner.

5. Experiments

We have implemented the two encoding schemes, and conducted two kinds of experiments: comparison with the existing encoding schemes for λ-terms [5, 12], and comparison with an encoding scheme for straight-line context-free (SLCF) tree grammars used in TreeRePair [7]. As mentioned in Section 1, since SLCF tree grammars may be encoded as λ-terms, higher-order data compression [6] may be viewed as a generalization of grammar-based compression [1, 7, 9]. The second experiment is motivated to check how much overhead is incurred by using the more expressive language of the λ-terms.

All the experiments reported below were performed on Intel Xeon CPU E3-1240 @ 3.40 GHz, with 16 gigabytes of RAM, and version 14.04 of the Ubuntu operating system. The tagless encoding scheme in Section 3 was implemented in Java, compiled with version 1.8.0_66 of Java SE Development Kit, and executed with Java Virtual Machine arguments -Xms16g, -Xmx16g, -Xmn3072m -Xss128m. The grammar-based encoding scheme in Section 4 was implemented in OCaml and compiled with version 4.01.0 of ocamlopt.

Tree data	ID	# Symbols	Alphabet
chromosome-1-3	DNA1	3000	5
chromosome-1-200	DNA2	200000	5
chromosome-2-200	DNA3	200000	5
chromosome-5-200	DNA4	200000	5
en-cancer-611	EN1	611	300
en-cancer-22788	EN2	22788	4423
en-arabidopsis-30594	EN3	30594	4710
cp.html	PROG1	10244	1170
fields.c	PROG2	3241	297
grammar.lsp	PROG3	1307	189
EXI-Array_Structural	XML1	226523	56
EXI-Telecomp_Structural	XML2	177634	49
EXI-factbook_Structural	XML3	55453	227
EXI-Invoice_Structural	XML4	15075	54
EXI-weblog_Structural	XML5	93435	56
1998statistics	XML6	28306	50

Table 1. Characteristics of the benchmarks. The column "Alphabet" shows the size of the alphabet.

5.1 Comparison with Previous Encoding Methods for λ-Terms

We have compared our encoding schemes with Tromp's one [12] and Vytiniotis and Kennedy's one [5]. For Tromp's encoding, we have implemented it in Java based on the paper's description. For Vytiniotis and Kennedy's encoding, we used the Haskell code based on their paper [5], which the first author kindly provided to us. The former was compiled and executed under the same Java environment as our tagless encoding, and the latter was compiled with version 7.6.3 of GHC.

As a benchmark set, we have used the λ-terms output by a higher-order compressor [4] for the string/tree data summarized in Table 1.

We used DNA sequences (composed of four types of bases 'A', 'G', 'C', and 'T') of Arabidopsis thaliana.[3] Each DNA sequence file named "chromosome-x-n" consists of the first $1000 \times n$ letters of chromosome x.[4] The data named "en-w-n" are English text files constructed from the abstracts of papers available at PubMed (`http://www.ncbi.nlm.nih.gov/pubmed`); we have collected the abstracts containing a certain keyword, and combined them into one text file. In "en-w-n", w stands for the keyword used for collecting the abstracts, and n stands for the number of words (including punctuations) in the data. The program source codes (PROG1-03) have been taken from the Canterbury Corpus (`http://corpus.canterbury.ac.nz`). The XML files (XML1–6) have been taken from the benchmark set of TreeRePair [7], which are originally from XMLCompBench (`http://xmlcompbench.sourceforge.net/Dataset.html`) and `http://www.cafeconleche.org/examples/`.

In the conversion of those data to λ-terms (using higher-order compression), we have treated English words and program tokens as just free variables. The names (i.e. the original words/tokens)

[3] available at www.arabidopsis.org

[4] The reason why the alphabet size is 5 rather than 4 is that, since the higher-order compressor takes a tree as an input, each sequence has been represented as a unary tree; we need a leaf symbol (that represents the end of a sequence) in addition to 'A', 'G', 'C', and 'T'.

ID	Term size	Tagless-	Tagless	Gram	G+opt	VK	VK+bzip2	Tromp	Tromp+bzip2
DNA1	2567	988	973	933	**905**	5221	1407	5464	1341
		(1.092)	(1.075)	(1.031)	(1.0)	(5.769)	(1.555)	(6.038)	(1.482)
DNA2	98042	59766	59307	54255	**53920**	-	-	4684189	65287
		(1.108)	(1.100)	(1.006)	(1.0)	-	-	(86.87)	(1.211)
DNA3	88316	53540	53484	48534	**48051**	-	-	3880905	58726
		(1.114)	(1.113)	(1.010)	(1.0)	-	-	(80.77)	(1.222)
DNA4	97727	59270	58809	53953	**53559**	-	-	4553082	64722
		(1.107)	(1.098)	(1.007)	(1.0)	-	-	(85.01)	(1.208)
EN1	1220	637	**636**	733	719	12308	901	11178	826
		(0.886)	(0.885)	(1.019)	(1.0)	(17.12)	(1.253)	(15.55)	(1.149)
EN2	40948	28591	28522	29287	**28136**	-	-	6891597	32168
		(1.016)	(1.014)	(1.041)	(1.0)	-	-	(244.9)	(1.143)
EN3	52800	37312	37232	37817	**36372**	-	-	10116151	42005
		(1.026)	(1.024)	(1.040)	(1.0)	-	-	(278.1)	(1.155)
PROG1	8241	5262	**5199**	6155	5214	-	-	418848	6224
		(1.009)	(0.997)	(1.180)	(1.0)	-	-	(80.33)	(1.194)
PROG2	4122	2214	2199	2358	**2167**	38831	2883	54697	2660
		(1.022)	(1.015)	(1.088)	(1.0)	(17.92)	(1.330)	(25.24)	(1.228)
PROG3	1641	807	**797**	933	848	8508	1178	13138	1053
		(0.952)	(0.940)	(1.100)	(1.0)	(10.03)	(1.389)	(15.49)	(1.242)
XML1	2236	988	970	986	**941**	4846	1291	11883	1226
		(1.050)	(1.031)	(1.048)	(1.0)	(5.150)	(1.372)	(12.63)	(1.303)
XML2	276	**123**	**123**	186	172	254	219	718	240
		(0.715)	(0.715)	(1.081)	(1.0)	(1.477)	(1.273)	(4.174)	(1.395)
XML3	3034	1676	1666	1773	**1597**	14166	2280	42929	2047
		(1.049)	(1.043)	(1.110)	(1.0)	(8.870)	(1.428)	(26.88)	(1.282)
XML4	223	**104**	**104**	189	168	263	197	548	215
		(0.619)	(0.619)	(1.125)	(1.0)	(1.565)	(1.173)	(3.262)	(1.280)
XML5	128	52	52	115	97	**45**	97	134	119
		(0.536)	(0.536)	(1.186)	(1.0)	(0.464)	(1.0)	(1.381)	(1.223)
XML6	1095	**449**	467	502	517	1264	693	4418	648
		(0.868)	(0.903)	(0.971)	(1.0)	(2.445)	(1.340)	(8.545)	(1.253)

Table 2. Comparison of encoders for (simply-typed) λ-terms in terms of the bit code size. The best values are in bold. '-' means that it did not terminate in 20 minutes. The numbers in parentheses show the normalized values, where the values for G+opt are set to 1.0.

of the free variables can be separately compressed using an off-the-shelf string compressor; since it is independent of the issue of encoding λ-terms, we have not counted the size of this part.

Table 2 shows the result of the experiments. The column "Term size" shows the sizes of λ-terms used as inputs, where the size $|M|$ of a λ-term M is defined by: $|n| = 1$, $|\lambda.M| = 1 + |M|$, and $|M_1 M_2| = |M_1| + |M_2| + 1$. The columns "Tagless-"/"Tagless" respectively report the results (the byte size of the bit code) for our tagless encoding without/with the optimization in Remark 3.2 in Section 3. The columns "Gram", and "G+opt" shows the result for our grammar-based encoding in Section 4; "Gram" is the basic one in Section 4.2, "G+opt" is the optimized one in Section 4.3, which takes β-equivalence into account. The columns "VK" and "Tromp" show the result for Vytiniotis and Kennedy's encoding [5] and Tromp's encoding [12] respectively. Since the bit codes produced by those encoding schemes were large, we have further compressed their bit codes with bzip2. The columns "VK+bzip2" and "Tromp+bzip2" show the size after applying bzip2.[5] Vytiniotis and Kennedy's encoder (taken from Section 5.2 of their paper [5]) was slow and did not terminate for large inputs.

We can observe that our encoding schemes clearly outperform the previous methods (even combined with bzip2) in terms of code size, with the only exception of XML5, for which the input λ-term is quite small.

[5] We have also tested adaptive range coding, but the result was worse than the combination with bzip2.

Among our encoding schemes, for large inputs (DNA2–4, EN2, and EN3), the grammar-based encoding works better (in terms of the code size) than the tagless one, even without the optimization, although the tagless encoding sometimes works better for smaller inputs. The latter is probably due to the overhead of keeping grammar information in the grammar-based encoding, which is outweighed by the benefit of the grammar-based encoding for larger inputs. To confirm this point, we show in Table 3 the size of the grammar information part and its ratio with respect to the total code size. We can observe that the grammar information dominates for small inputs, but for larger inputs, it accounts for only a few percent of the total code size.

As for the basic vs optimized versions of the grammar-based encoding, the optimized version consistently works better in terms of the code size, although the gain is at most a few percent.

Table 4 shows running times of our encoders. The grammar-based encoding is much slower than the tagless one. We, however, think that it is due to the naiveness of the current implementation, not a fundamental limitation of the approach. The optimization of the implementation is left for future work.

5.2 Comparison with an Encoder for SLCF Tree Grammars

We have also compared our encoding scheme with a bit encoder of TreeRePair [7] for SLCF tree grammars. TreeRePair is a grammar-based compressor for tree-structured data. It first converts tree data

	Gram			G+opt		
ID	Total	Gram.	(%)	Total	Gram.	(%)
DNA1	933	77	(8)	905	62	(7)
DNA2	54255	152	(0.3)	53920	103	(0.2)
DNA3	48534	361	(0.7)	48051	174	(0.4)
DNA4	53953	229	(0.4)	53559	126	(0.2)
EN1	733	112	(15)	719	99	(14)
EN2	29287	1481	(5)	28136	589	(2)
EN3	37817	1736	(5)	36372	658	(2)
PROG1	6155	1467	(24)	5214	615	(12)
PROG2	2358	316	(13)	2167	169	(8)
PROG3	933	183	(20)	848	111	(13)
XML1	986	102	(10)	941	77	(8)
XML2	186	79	(42)	172	68	(40)
XML3	1773	307	(17)	1597	171	(11)
XML4	189	99	(52)	168	79	(47)
XML5	115	78	(68)	97	61	(63)
XML6	502	100	(20)	517	124	(24)

Table 3. The details on the codes produced by our grammar-based scheme. The numbers in parentheses show the ratio of the grammar size to the total code size.

ID	Term size	Tagless	Gram	G+opt
DNA1	2567	0.01	0.05	0.05
DNA2	98042	0.07	102	98
DNA3	88316	0.03	87	77
DNA4	97727	0.03	98	91
EN1	1220	0.0007	0.07	0.07
EN2	40948	0.008	92	85
EN3	52800	0.009	159	150
PROG1	8241	0.004	5	5
PROG2	4122	0.001	0.5	0.5
PROG3	1641	0.0006	0.07	0.07
XML1	2236	0.0005	0.06	0.06
XML2	276	0.0003	0.006	0.006
XML3	3034	0.001	0.3	0.3
XML4	223	0.0003	0.006	0.005
XML5	128	0.0003	0.003	0.002
XML6	1095	0.01	0.03	0.03

Table 4. The times (in seconds) spent for encoding.

into an SLCF tree grammar, which consists of the production rules of the form:

$$\begin{array}{l} A_0\ x_0\ \cdots\ x_{n_0} \to t_0 \\ \cdots \\ A_{m-1}\ x_1\ \cdots\ x_{n_{m-1}} \to t_{m-1} \\ S \to t_m. \end{array}$$

Here each non-terminal A_i takes tree parameters $x_1, \ldots, x_{n_i}$, and the righthand side t_i consists of tree constructors, variables, and non-terminals A_j's such that $j < i$. TreeRePair then converts the SLCF grammar to a bit code.

The SLCF grammar above can be easily represented as the following λ-term.

$$\begin{array}{l} \mathbf{let}\ A_0 = \lambda x_1 : \mathtt{o}. \cdots . \lambda x_{n_0} : \mathtt{o}. t_0\ \mathbf{in} \\ \cdots \\ \mathbf{let}\ A_{m-1} = \lambda x_1 : \mathtt{o}. \cdots . \lambda x_{n_{m-1}} : \mathtt{o}. t_{m-1}\ \mathbf{in} \\ \quad t_m. \end{array}$$

We have converted the SLCF grammars produced by TreeRePair to λ-terms, and then compared the size of the bit codes obtained from the λ-terms by using our encodings, with the size of the bit codes produced from the SLCF grammars by the bit encoder part of TreeRePair. Obviously, this experiment is favorable to TreeRePair, since TreeRePair assumes the structure of SLCF grammars, while our encoding does not; for example, for the lefthand side of each production rule, TreeRePair only needs to record the number of the variables (as all the arguments are restricted to trees in SLCF grammars), but our encoder needs to record the type of each variable. The goal of the experiments is to check how much we can reduce the overhead incurred by the use of λ-terms.

Table 5 shows the result. As the benchmark data, we have reused the string/tree data in Table 1, and used TreeRePair to convert them to SLCF tree grammars. The SLCF tree grammars have further been converted to λ-terms for our encoders. The column "Term size" shows the sizes of the λ-terms; they do not coincide with those in Table 2, as the former has been obtained from the SLCF grammars, while the latter has been produced by the higher-order compressor. The source code of TreeRePair was taken from `code.google.com/p/treerepair/`, and compiled with version 4.8.4 of GCC.

As observed in the table, the optimized version of our grammar-based encoding works almost as well as the encoder of TreeRePair, despite that the latter is specialized for SLCF grammars; sometimes ours even outperforms TreeRePair. The tagless encoding is not good enough; it is clearly inferior to TreeRePair (and the grammar-based encoders) for DNA1–4. For EN1–3 and PROG1–3, the tagless encoding actually outperforms TreeRePair. This is probably due to the difference between arithmetic coding and Huffman coding; TreeRePair uses Huffman encoding for symbols, which may not work well when the size of the alphabet is large.

		Tagless	Gram	G+opt	TRePair
ID	Term size	Size	Size	Size	Size
DNA1	2559	964	919	897	**873**
DNA2	97567	56864	53204	52964	**52819**
DNA3	89093	52294	48502	**48242**	48741
DNA4	97407	56958	53209	**52953**	52958
EN1	1219	740	695	**682**	1002
EN2	41167	28235	27918	**27711**	35010
EN3	53073	36832	**36253**	35966	43968
PROG1	8895	5355	**5046**	5484	6611
PROG2	4133	2264	**2107**	2171	2439
PROG3	1645	863	**823**	853	1003
XML1	7492	3406	3108	3100	**3067**
XML2	2444	1000	953	926	**904**
XML3	3211	1737	**1557**	1615	1630
XML4	539	**237**	258	259	250
XML5	2030	800	733	720	**690**
XML6	1127	467	491	485	**429**

Table 5. Comparison with TreeRePair. The column "Size" shows the code size in bytes, and the best values are in bold. The column "TRePair" shows the result for TreeRePair.

6. Related Work

The issue of compact bit encoding of λ-terms has not been attracting much attention to our knowledge. As mentioned in Section 1, Tromp [12] proposed a bit encoding scheme for untyped λ-terms, but his encoding was not so compact. Vytiniotis and Kennedy [5] proposed another bit encoding scheme, where a typed λ-term is represented as a sequence of yes/no answers to questions for identifying the λ-term. As reported in Section 5, their encoding is not so compact either. Compared with these previous encodings, we utilize type information more aggressively. In our second scheme,

β-equivalence has also been taken into account for further optimizations. Vytiniotis and Kennedy [5] briefly discussed the idea of combining their game-based encoding with arithmetic coding, but it was left for future work.

Our grammar-based coding scheme uses the idea of Cameron's grammar-based arithmetic coding [2]. Actually, he applied his encoding scheme to compression of Pascal programs. Our key insight in the grammar-based coding scheme with respect to his work is that, for the simply-typed λ-calculus, the typing rules may be viewed as production rules of a context-free grammar. Although the resulting grammar is infinite (i.e. consists of infinitely many production rules), for a given λ-term, we only need finitely many instances of the typing rules. This idea of viewing a type system as a part of a language grammar may be useful also for compression of typed functional programs.

As mentioned in Section 1, this work has been motivated by higher-order compression [6], which may be viewed as a generalization of grammar-based compression [1, 7, 9]. Since grammar-based compression represents data as straight-line programs that generate the data, there is also an issue on how to represent the grammars as bit sequences. As the simply-typed λ-terms are strictly more expressive than straight-line grammars (the latter can be easily represented by the former, but not vice versa), the compact bit encoding of λ-terms is more challenging. According to our experiments in Section 6, our encoding is nearly as compact as that of straight-line context-free tree grammars in TreeRePair [7], despite the expressive power of the λ-terms.

7. Conclusion

We have proposed two type-based bit encoding schemes for simply-typed λ-terms. We have implemented the schemes and confirmed that they outperform previous encoding schemes for λ-terms. We have so far checked the effectiveness of our coding scheme only in the context of higher-order compression; evaluation of the effectiveness in other contexts is left for future work. Future work also includes an improvement of the efficiency of the grammar-based coding.

Acknowledgment

We would like to thank anonymous referees for a number of useful comments. This work was supported by JSPS Kakenhi 15H05706 and 23220001.

Appendix

A. More Detail on Cameron's Encoding Scheme

We describe below a little more detail on Cameron's encoding/decoding scheme reviewed in Section 4.1.

Suppose that a grammar $\mathcal{G}$ consists of the set of production rules:

$$\{A_i \to X_{i,j,1} \cdots X_{i,j,k_{i,j}} \mid i \in \{1, \ldots, m\}, j \in \{1, \ldots, n_i\}\},$$

and that $p_{i,j} (> 0)$ is the probability that the production $A_i \to X_{i,j,1} \cdots X_{i,j,k_{i,j}}$ is used in rewriting A_i, where $\Sigma_{j \in \{1,\ldots,n_i\}} p_{i,j} = 1$ for each $i \in \{1, \ldots, m\}$.

Let $r_{i,j} = \Sigma_{j' \in \{1,\ldots,j\}} p_{i,j'}$. We assign the "range" $[r_{i,j-1}, r_{i,j})$ to the rule $A_i \to X_{i,j,1} \cdots X_{i,j,k_{i,j}}$. The range can be lifted to that of a production sequence as follows.

$$\begin{array}{l} \mathit{Range}_{\mathcal{G}}(\epsilon) = [0, 1) \\ \mathit{Range}_{\mathcal{G}}([A_i \to X_{i,j,1} \cdots X_{i,j,k_{i,j}}]\pi) = \\ \quad \textbf{let } [x, y) = \mathit{Range}_{\mathcal{G}}(\pi) \textbf{ in} \\ \quad\quad [r_{i,j-1} + x p_{i,j}, r_{i,j-1} + y p_{i,j}). \end{array}$$

Intuitively, the range of $[A_i \to X_{i,j,1} \cdots X_{i,j,k_{i,j}}]\pi$ is obtained by choosing a subrange of the range $[r_{i,j-1}, r_{i,j-1} + p_{i,j})$ of $[A_i \to X_{i,j,1} \cdots X_{i,j,k_{i,j}}]$, according to π.

Let $\mathit{RangeToCode}(I)$ be a function that returns a bit sequence $b_1 \cdots b_k$ such that

$$[\Sigma_{i \in \{1,\ldots,k\}} 2^{-i} b_i, 2^{-k} + \Sigma_{i \in \{1,\ldots,k\}} 2^{-i} b_i) \subseteq I.$$

Then, the encoding function for Cameron's scheme can be defined by:

$$\mathit{EncC}_{\mathcal{G}}(w) = \mathit{RangeToCode}(\mathit{Range}_{\mathcal{G}}(\mathit{Parse}_{\mathcal{G}}(w))).$$

Here, $\mathit{Parse}_{\mathcal{G}}$ is a parsing function that converts a word to the corresponding leftmost derivation sequence. For the sake of simplicity, we have assumed that arbitrary-precision arithmetic is available, ignoring the efficiency issue; in the actual coding, fixed-precision arithmetic is used, and the bit code is incrementally output, instead of directly computing $\mathit{Range}_{\mathcal{G}}(\mathit{Parse}_{\mathcal{G}}(W))$.

For decoding, we first define a function $\mathit{NumToSeq}_{\mathcal{G}}$ for converting a number $r \in [0, 1)$ to the derivation sequence π such that $r \in \mathit{Range}_{\mathcal{G}}(\pi)$.

$$\begin{array}{l} \mathit{NumToSeq}_{\mathcal{G}}(r) = \mathit{NumToSeqAux}_{\mathcal{G}}(r, A_1) \\ \mathit{NumToSeqAux}_{\mathcal{G}}(r, \epsilon) = \epsilon \\ \mathit{NumToSeqAux}_{\mathcal{G}}(r, A_i W) = \\ \quad [A_i \to X_{i,j,1} \cdots X_{i,j,k_{i,j}}] \\ \quad \mathit{NumToSeqAux}_{\mathcal{G}}((r - r_{i,j-1})/p_{i,j}, X_{i,j,1} \cdots X_{i,j,k_{i,j}} W) \\ \quad \text{where } r \in [r_{i,j-1}, r_{i,j}) \\ \mathit{NumToSeqAux}_{\mathcal{G}}(r, aW) = \mathit{NumToSeqAux}_{\mathcal{G}}(r, W) \\ \quad\quad \text{(if } a \text{ is a terminal)} \end{array}$$

The decoding function $\mathit{DecC}_{\mathcal{G}}$ is then defined by:

$$\mathit{DecC}_{\mathcal{G}}(s) = \mathit{Unparse}_{\mathcal{G}}(\mathit{NumToSeq}_{\mathcal{G}}(\mathit{ToNum}(s))),$$

where $\mathit{Unparse}_{\mathcal{G}}$ is the inverse of $\mathit{Parse}_{\mathcal{G}}$ and $\mathit{ToNum}(b_1 \cdots b_k) = \Sigma_{i \in \{1,\ldots,k\}} 2^{-i} b_i$.

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] Robert D Cameron. Source encoding using syntactic information source models. *Information Theory, IEEE Transactions on*, 34(4):843–850, 1988.

[3] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[4] K. Matsuda K. Takeda, N. Kobayashi. Optimization of a repair-based higher-order com- pression algorithm. *Proc. 31th JSSST 2014*, 31:346–359, 2014. (in Japanese).

[5] Andrew J. Kennedy and Dimitrios Vytiniotis. Every bit counts: The binary representation of typed data and programs. *J. Funct. Program.*, 22(4-5):529–573, 2012.

[6] Naoki Kobayashi, Kazutaka Matsuda, Ayumi Shinohara, and Kazuya Yaguchi. Functional programs as compressed data. *Higher-Order and Symbolic Computation*, 25(1):39–84, 2012.

[7] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. Tree structure compression with repair. In *Data Compression Conference (DCC), 2011*, pages 353–362. IEEE, 2011.

[8] G Nigel N Martin. Range encoding: an algorithm for removing redundancy from a digitised message. In *Proc. Institution of Electronic and Radio Engineers International Conference on Video and Data Recording*, 1979.

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

[10] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

[11] Julian Seward. bzip2 : Home. `http://bzip.org`.

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

[13] Dimitrios Vytiniotis and Andrew J. Kennedy. Functional pearl: every bit counts. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 15–26. ACM, 2010.

[14] Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.