Compaction of Church Numerals for Higher-Order Compression

Isamu Furuya\textsuperscript{1}  Takuya Kida\textsuperscript{1}
\textsuperscript{1}Graduate School of IST, Hokkaido University, Japan
\{furuya,kida\}@ist.hokudai.ac.jp

Abstract
We addressed the problem of compacting the Church numerals, which is useful for higher-order compression. We proposed a novel decomposition scheme for a natural number using tetration, which leads to the compact representation of $\lambda$-terms equivalent to the Church numerals. For natural number $n$, we proved that the size of the lambda term obtained by our method is $O(s\log\log n \log\log n)$. We also quantitatively confirmed via experiments that in many cases, our method can produce shorter $\lambda$-terms more than existing methods.

1 Introduction

Our aim is to obtain a compact lambda term ($\lambda$-term) equivalent to the Church numeral for a given natural number. Church numerals are unary representations of natural numbers using lambda notation; for an integer $n$, the length of the Church numeral linearly increases with $n$. Let $C(n)$ be the Church numeral for a natural number $n$; then the lambda expression $C(n)$ is as follows:

$$C(n) = (\lambda fx. f (\cdots f (f x) \cdots)).$$

For a large number $n$, decomposing it and representing its expression may reduce the length of the $\lambda$-term for $n$. For example, let $n = 50$; we can decompose it as $2 \times 5 \times 5$. The $\lambda$-term corresponding to the expression is $(\lambda pqfx.p (q f) x) C(2)((\lambda pfx.p (p f) x) C(5))$, which is much shorter than $C(50)$.

Length shortening of Church numerals is applied to data compression. Kobayashi et al.\textsuperscript{2} proposed a compression method, called higher-order compression, which uses a $\lambda$-term as the data model. The method translates an input into a $\lambda$-term inducing the input itself, and then encodes the obtained $\lambda$-term. The input data is first represented as a simple $\lambda$-term, which consists of nested functional applications of lambda notation. The $\lambda$-term is converted into a shorter term by repeatedly extracting common contexts. Since repeating patterns in the initial $\lambda$-term appear as Church numerals, quickly shortening them is important for compression of data. We call the task of shortening Church numerals a compaction of $C(n)$.

In this paper, we introduce a new scheme, named Recursive tetranal partitioning (RTP) to decompose a natural number using tetration. We present an algorithm to perform RTP for a given natural number and show that the
expression obtained by RTP is translated into a compact $\lambda$-term. Moreover, we prove that the length of the obtained $\lambda$-term is $O(s\log_2 n \log n / \log \log n)$ in the worst case.

Kobayashi et al. [1] also showed the binary representation of $\lambda$-terms equivalent to Church numerals, whose length is $O(\log n)$. Although the result of our $O(s\log_2 n \log n / \log \log n)$ is larger than $O(\log n)$, we confirmed via an experiment that the obtained $\lambda$-terms become much shorter than binary representation in many cases.

Yaguchi et al. [4] recently proposed an efficient algorithm for higher-order compression. They utilized a simply typed $\lambda$-term for efficient modeling and encoding. Their idea was to extract the most frequent context up to a certain size, unlike the approach of Kobayashi et al., which extracts every context occurring more than once. In [4], they state that the performance of their method is often better than the performance of grammar compression, with regard to compression ratio. We also confirmed that our method tends to produce more compact $\lambda$-terms for highly repeating patterns compared with the method of Yaguchi et al.. Our method can be easily incorporated into their algorithm.

Contributions: Three contributions are made in this paper:

1. For large natural numbers, we propose a novel decomposition scheme using tetration, RTP, which leads to the compact representation of $\lambda$-terms equivalent to the Church numerals of the numbers. RTP is different from the $n$-ary notation.

2. By incorporating RTP, we present an algorithm performing the compaction of $C(n)$. Moreover, we proved that the length of $\lambda$-terms, constructed by the algorithm, is $O(s\log_2 n \log n / \log \log n)$ in the worst case.

3. We quantitatively confirmed via experiments that in many cases, our method can produce shorter $\lambda$-terms more than the existing methods; the size reduction is approximately 21% and 23% for $n \leq 10,000$ on an average, compared with the binary representation stated in [1] and with the method of Yaguchi et al. stated in [4], respectively.

The rest of this paper is organized as follows. In the next section, we review the lambda notation, Church numerals, and tetration. In Sec. 3 we define RTP and present the translation algorithm using RTP. Moreover, in Sec. 4 we prove the upper bound of the length of the $\lambda$-term produced by the algorithm. In Sec. 4 we discuss how to apply our algorithm to higher-order compression and illustrate an experimental result. Finally, we conclude the paper in Sec. 5.

2 Preliminary

2.1 Lambda terms

Definition 1 (Lambda terms). Let $S = \{\lambda, \_, (\_), \}\}$ be the set of special symbols. Let $A$ be the set of characters in the input data, where we assume $A \cap S = \emptyset$. We call $A$ and $a \in A$ terminal alphabet and terminal symbol, respectively. Let $\Sigma$ be an alphabet such that $\Sigma \cap (A \cup S) = \emptyset$. We call $x \in \Sigma$ variable. For $a \in A$ and $x \in \Sigma$, lambda terms ($\lambda$-terms) are recursively defined as follows:

(i) $x$  
(ii) $(\lambda x.M)$  
(iii) $(M \ N)$  
(iv) $a$
where $M$ and $N$ are $\lambda$-terms.

We call (ii) in Def. II $\lambda$-abstraction and (iii) functional application. Although condition (iv) is added for higher-order compression, Def. II is inherently the same as that of the lambda calculus. Thus, hereafter, we borrow well-known omission rules of lambda calculus, such as the omission of parentheses and short notation of nested $\lambda$-abstractions. We also follow the usual rules of lambda terms. We define the size of a $\lambda$-term according to II as follows.

**Definition 2** (The size of a lambda term). We denote the size of the $\lambda$-term $M$ by $\# M$, and define it as follows:

\[
\begin{align*}
\# x &= \# a = 1, \\
\# \lambda x . M &= \# M + 1, \\
\# (M \, N) &= \# M + \# N + 1,
\end{align*}
\]

where $a \in A, x \in \Sigma$, and $M, N$ are $\lambda$-terms.

**2.2 Church numerals**

Natural numbers are known to be represented by $\lambda$-terms as Church numerals.

**Definition 3** (Church numerals). Let $n$ be a natural number. Church numerals for $n$, denoted by $C(n)$, are defined as follows:

\[
C(n) := (\lambda fx. \underbrace{n \cdot \ldots \cdot n}_n \, f \, x).
\]

For example, $C(5)$ is $(\lambda fx. f(f(f(f(f x)))))$. From the following definition, it is also known that some arithmetic operations on numbers can be represented by functions of $\lambda$-terms on Church numerals.

**Definition 4** (Arithmetic operations on Church numerals). Let $n_1$ and $n_2$ be natural numbers. From Def. 3, each function of addition, multiplication, and exponentiation on $C(n_1)$ and $C(n_2)$ are, respectively, defined as follows:

\[
\begin{align*}
\text{addition} \quad \text{add}(n_1, n_2) &= n_1 + n_2 := (\lambda pq f x. p \, (q \, f \, x)) \, C(n_1) \, C(n_2), \\
\text{multiplication} \quad \text{mul}(n_1, n_2) &= n_1 \cdot n_2 := (\lambda pq f x. (q \, f) \, x) \, C(n_1) \, C(n_2), \\
\text{exponentiation} \quad \text{exp}(n_1, n_2) &= n_1^{n_2} := (\lambda pq f x. q \, p \, f \, x) \, C(n_1) \, C(n_2).
\end{align*}
\]

As can be seen, $\lambda$-abstractions at the above $\lambda$-terms appear first and are followed by Church numerals. We call the former function parts and the latter argument parts.

**2.3 Tetration and super-logarithm**

Tetration is known as the next hyper-operation after exponentiation. In mathematics, it is defined as iterated exponentiation. For any natural numbers $\varphi$ and $i$, the $i$th tetration of $\varphi$ is denoted by $\varphi^i$ and recursively defined as follows:

\[
\varphi^i := \begin{cases} 
1, & \text{for } i = 0, \\
\varphi^{i-1} \varphi, & \text{for } i > 0.
\end{cases}
\]

For example, $1^2 = 2$, $2^2 = 4$, $3^2 = 16$, and $4^2 = 65536$.

The following lemma is easily induced from the definition.
Lemma 1. For natural numbers $\varphi$ and $i$, it holds that $\log_{\varphi}^{i} \varphi = i - 1$.

The super-logarithm, denoted by $\text{slog}$, is known as one of the inverse operations of tetration. For natural numbers $\varphi$ and $i$, it holds that $\text{slog}_{\varphi}^{i} \varphi = i$. On the other hand, the iterated logarithm of $n$, denoted by $\log^{*} n$, is also known as the number of times the logarithm function must be applied to $n$ before it becomes less than or equal to 1. For the positive numbers, the super-logarithm is essentially equivalent to the iterated logarithm; it holds that $\log^{*} n = \lceil \text{slog}_{\varphi} n \rceil$ for any $n > 0$.

3 Proposed Method

3.1 Our approach

We conduct the compaction of $C(n)$ in the following two steps:

**Step 1:** Decompose $n$ with a natural number $\varphi$ ($1 < \varphi < n$) into a numerical expression, which includes as much tetration and multiplication of $\varphi$ as possible.

**Step 2:** Translate the expression into a corresponding $\lambda$-term so that the translated $\lambda$-term includes a single functional part followed by $C(\varphi)$.

For Step 1, we introduce RTP in Sec. 3.2. For Step 2, we present a translation algorithm in Sec. 3.3.

As stated in Sec. 1, we may reduce $\#C(n)$ for a large number $n$. For the running example, it becomes $\#C(50) = 103$, while $\#(\lambda p q f x.p(q f)x)C(2)(\lambda p f x.p(p f)x)C(5) = 44$ corresponding to $2 \times 5 \times 5$. Moreover, we can compress the $\lambda$-term by combining two function parts into one, such as $(\lambda p q f x.p(q(q f))x) C(2) C(5)$ with a size of 35.

For a natural number $n$, there are many ways of decomposition, and the size of the $\lambda$-term changes depending on the approach followed. For the running example, expressions $7 \times 7 + 1$ and $5 \times (5 + 5)$ are also equal to 50. The corresponding $\lambda$-terms are

$7 \times 7 + 1 : (\lambda p f x.p(p f))(f x) \ C(7)$,  
$5 \times (5 + 5) : (\lambda p f x.p(p(f p f))x) \ C(5)$,

and the sizes are 29 and 28, respectively. Obtaining the optimal decomposition is difficult, therefore, in RTP, we employ a heuristic approach.

3.2 Recursive tetrational partitioning

We only consider numerical expressions such as

$$F ::= x \mid F + F \mid F \cdot F \mid F^F$$

in BNF, where $x$ is an arbitrary natural number. If the calculation of $F$ results in $n$, we denote it by $F[n]$.

We construct a compact $F[n]$ such that the size of the $\lambda$-term is reduced. It is effective in reducing Church numerals appearing in the $\lambda$-term. It is equal to reduce the kind of natural numbers included in $F[n]$.  

4
Let φ be a natural number smaller than or equal to n. Then, we consider numerical expressions only using φ, denoted by \( F_\varphi \), as follows:

\[
F_\varphi := \varphi \mid F_\varphi + F_\varphi \mid F_\varphi \cdot F_\varphi \mid F_\varphi^{F_\varphi}.
\]

\( F_\varphi \) derives only a multiple of \( \varphi \). Let \( r = n \mod \varphi \) and \( \bar{n} = n - r \). Then, if \( F_\varphi \) derives \( \bar{n} \), we denote it by \( F_\varphi[\bar{n}] \). Here, \( F_\varphi[\bar{n}] + r \) is a numerical expression which derives \( n \) and includes at most 2 natural numbers, \( \varphi \) and \( r \).

In order to reduce the size of the \( \lambda \)-term, it is also effective in reducing arithmetic operations appearing in \( F_\varphi[\bar{n}] \), because the size of the \( \lambda \)-term increases with the number of arithmetic operations. Next we show how our method achieves this reduction.

We partition \( \bar{n} \) into an addition of tetrations with integer coefficients. It is denoted as follows:

\[
\bar{n} \rightarrow k_\varphi \cdot p_k + k_{-1} \cdot p_{k-1} + \cdots + 1 \cdot p_1
\]

where \( k \) is the maximum natural number such that \( k \varphi \leq \bar{n} \) and \( p_i \) (\( 0 \leq i \leq k \)) is the integer such that \( 0 \leq p_i < i+1 \varphi \). The term including \( 0 \varphi \) does not appear in it because \( \bar{n} \) is divisible by \( \varphi \). Then, we convert each term as follows:

\[
i \varphi \cdot p_i \rightarrow i \varphi \cdot (p_i - r_i) + (\underbrace{i \varphi + \cdots + i \varphi}_{r_i \varphi})
\]

where \( r_i = p_i \mod \varphi \). Moreover, let \( \bar{p}_i = p_i - r_i \) and we partition \( \bar{p}_i \) recursively in the same way. In this way, we can convert \( \bar{n} \) into \( F_\varphi[\bar{n}] \). The procedure above can be defined as Def. 5.

**Definition 5** (Recursive tetrahedral partitioning (RTP)). Let \( n \) and \( \varphi \) be natural numbers such that \( 0 < \varphi < n \), and let \( r = n \mod \varphi \) and \( \bar{n} = n - r \). Then, we define RTP as follows, with the result derived by RTP being denoted by \( T_\varphi[\bar{n}] \):

\[
T_\varphi[\bar{n}] = \begin{cases} 
\bar{n} & \text{(if } \bar{n} \leq \varphi) \\
 k_\varphi \cdot (T_\varphi[p_k - (p_k \mod \varphi)]) + (\underbrace{k_\varphi + \cdots + k_\varphi}_{p_k \mod \varphi}) \\
+ \cdots + 1 \varphi \cdot (T_\varphi[p_1 - (p_1 \mod \varphi)]) + (\underbrace{1 \varphi + \cdots + 1 \varphi}_{p_1 \mod \varphi}) & \text{(otherwise)}
\end{cases}
\]

where \( k \) is the maximum natural number such that \( k \varphi \leq \bar{n} \) and \( p_i \) (\( 0 \leq i \leq k \)) is the integer such that \( 0 \leq p_i < i+1 \varphi \). Here, if \( p_i = 0 \) or \( 1 \), we omit the display of the term or coefficient, respectively.

For example, \( T_\varphi[66579] = T_\varphi[65578] = 4^2 + 3^2 \times 2 + 2^2 \times 2 + 2 \) with \( n = 65579 \) and \( \varphi = 2 \). In Def. 5 each coefficient \( (p_i - (p_i \mod \varphi)) \) is a multiple of \( \varphi \). Thus, the remainder of each recursion step will always be 0. Hence, there is no term including \( 0 \varphi \) in \( T_\varphi[\bar{n}] \). With regard to given \( n \) and \( \varphi \), \( T_n[\varphi] \) is uniquely determined.

### 3.3 Translation algorithm

When a numeral expression \( F \) is represented by Polish notation, we denote it by \( PN(F) \). Then, the following holds:

\[
PN(T_\varphi[\bar{n}]) := +(\text{term}(k), +(\text{term}(k-1), (\cdots, +(\text{term}(2), \text{term}(1)))\cdots),
\]

5
Algorithm 1 Algorithm translating $T_\varphi[\bar{n}] + r$ into a $\lambda$-term

Input: $T_\varphi[\bar{n}], r$
Output: $\Lambda(T_\varphi[\bar{n}], r)$

1: function Translate($F$)
2:   function TermCase($F$)
3:     if $F = "\times(M,N)"$ then return Translate($F$)
4:     else return (Translate($F$) $f$)
5:   end if
6: end function
7: if $F = +(M,N)$ then return ((TermCase($M$))($\text{TermCase}(N)$))
8: else if $F = "\times(M,N)"$ then return ((Translate($M$))($\text{Translate}(N)$) $f$)
9: else if $F = "\wedge(M,N)"$ then return ((\text{Translate}(N))($\text{Translate}(M)$))
10: else return $p$
11: end if
12: end function
13: main
14: $PN(T_\varphi[\bar{n}]) ← T_\varphi[\bar{n}]$ denoted by Polish notation
15: output $(\lambda pfx.(\text{Translate}(PN(T_\varphi[\bar{n}])))(f f f f f f f f))$ $C(\varphi)$
16: end main

where

$$\text{term}(i) := +(\times(i \varphi, (\text{PN}(T_\varphi[p_i - (p_i \mod \varphi)]))))), \text{rem}(p_i)),$$

$$i \varphi := \wedge(\varphi, \cdots \wedge (\varphi, \cdots) \cdots), \text{and} \text{rem}(p_i) := +(i \varphi, +(i \varphi, \cdots + (i \varphi, i \varphi) \cdots).$$

For example,

$$PN(T_2[65578]) = PN(4^2 + 32 \times 2 + 2^2 \times 2 + 2)$$

$$= +(\wedge(2, \wedge(2, \wedge(2, \cdots) \cdots), +(\times(\wedge(2, \wedge(2, \wedge(2, \wedge(2, \cdots) \cdots, +(\times(\wedge(2, \wedge(2, \wedge(2, \cdots) \cdots, (2)).$$

In Sec. 3.1 we mentioned that any numerical expression can be represented on $\lambda$-terms by using Polish notation and naive substitution via Def. 4. However, the $\lambda$-term generated in this way tends to be large. Therefore, we design an algorithm, which enables the generation of a compact $\lambda$-term with regard to $T_\varphi[\bar{n}]$. This algorithm is presented in Algorithm 1. For example, by this algorithm, $65579 = T_2[65578] + 1$ is translated to $\Lambda(T_2[65578], 1)$ such that

$$\Lambda(T_2[65578], 1) = (\lambda pfx.p p f (p p p (p f))(p p f (p f))(f x)) C(2).$$

Here, $\#\Lambda(T_2[65578], 1) = 46$ is much smaller than $\#C(65579) = 131161$.

The $\lambda$-term generated by Algorithm 1 comprises only a $\lambda$-abstraction and $C(\varphi)$. Here, the $\lambda$-abstraction is regarded as a folded function of arithmetic operations included in $T_\varphi[\bar{n}]$. We denote the $\lambda$-term by $\Lambda(T_\varphi[\bar{n}], r)$ with $T_\varphi[\bar{n}]+r$, the numerical expression of $n$. 6
Lemma 2. Let $n$ and $\varphi$ be natural numbers such that $\varphi < n$, and let $r = n \mod \varphi$ and $\bar{n} = n - r$. Let us denote the number of additions, multiplications, and expressions appearing in $T_{\varphi}[\bar{n}]$ by $N_a$, $N_m$, and $N_e$, respectively. Then, the following equation holds:

$$\# \Lambda(T_{\varphi}[\bar{n}], r) = 2((2N_a + N_m + N_e) + \varphi + r + 6).$$

Proof. The form of $\Lambda(T_{\varphi}[\bar{n}], r)$ is $(\lambda n f . M) \mathcal{C}(\varphi)$. By Def. 2 and Def. 3, $\# \mathcal{C}(\varphi) = 2\varphi + 3$. Then, the size of that $\lambda$-term is as follows:

$$\# \Lambda(T_{\varphi}[\bar{n}], r) = \#(\lambda n f . M) \mathcal{C}(\varphi) = 3 + \# M + \# \mathcal{C}(\varphi) + 1 = \# M + 2\varphi + 7.$$

(1)

Initially, the $\lambda$-term $M$ has the form such as $(p f X)$ with $X = (f \ (f \cdots (f \ x) \cdots))$. Here, $\# M = 2r + 5$. If an addition is added in the $\lambda$-term, the form of $M$ changes into $(p f (p f X))$, and $\# M$ increases by 4. Note that while counting the size of the $\lambda$-term, the parentheses are ignored. Similarly, in the case of multiplications and exponentiations, the form of $M$ changes into $(p (p f) X)$ and $(p p f X)$, respectively. Both of their sizes increase by 2. Therefore, $\# M = 4N_a + 2N_m + 2N_e + 2r + 5$. Then, by equation (1), the following holds:

$$\# \Lambda(T_{\varphi}[\bar{n}], r) = \# M + 2\varphi + 7 = 4N_a + 2N_m + 2N_e + 2r + 5 + 2\varphi + 7 = 2((2N_a + N_m + N_e) + \varphi + r + 6). \square$$

Lemma 3. Let $n$ and $\varphi$ be natural numbers such that $\varphi < n$, and let $r = n \mod \varphi$ and $\bar{n} = n - r$. Then, if $n \leq 8$, then $\# \mathcal{C}(n) \leq \# \Lambda(T_{\varphi}[\bar{n}], r)$.

Proof. By Lemma 2, the following inequality holds:

$$\# \Lambda(T_{\varphi}[\bar{n}], r) = 2((2N_a + N_m + N_e) + \varphi + (n \mod \varphi) + 6) \geq 2(1 + \varphi + 6) \geq 2(1 + 2 + 6) = 18$$

If $n \leq 7$, then $\# \mathcal{C}(n) = 2n + 3 < 18$. Even if $n = 8$, then $\# \Lambda(T_{\varphi}[\bar{n}], r) > \# \mathcal{C}(8) = 19$ because $\# \Lambda(T_{\varphi}[8], 0) = 20$, $\# \Lambda(T_{\varphi}[8], 2) = 26$, and $\# \Lambda(T_{\varphi}[8], 0) = 24$. \square

3.4 Further compaction

We denote $\varphi$ in minimum $\# \Lambda(T_{\varphi}[\bar{n}], r)$ by $\varphi^*$, and denote $T_{\varphi^*}[\bar{n}]$ by $T^*[\bar{n}]$.

Lemma 4. With any natural number $n$, $\varphi^*$ is in $[2, 2\sqrt{n}]$.

Proof. Let $N_a(T_{\varphi}[\bar{n}])$, $N_m(T_{\varphi}[\bar{n}])$, and $N_e(T_{\varphi}[\bar{n}])$ be the numbers of additions, multiplications, and exponentiations in $T_{\varphi}[\bar{n}]$, respectively. Note that we can generate $T_{\varphi}[\bar{n}]$ with some arithmetic operations only if $\varphi \leq n/2$. Let $\varphi_t$ be an integer such that $\sqrt{n} < \varphi_t \leq n/2$, and let $r_{\varphi_t} = n \mod \varphi_t$ and $\bar{n}_{\varphi_t} = n - r_{\varphi_t}$. Here, $T_{\varphi_t}[\bar{n}_{\varphi_t}]$ includes some additions, no multiplication and no exponentiation.

Thus, $N_a(T_{\varphi_t}[\bar{n}_{\varphi_t}]) \geq 1$, $N_m(T_{\varphi_t}[\bar{n}_{\varphi_t}]) = N_e(T_{\varphi_t}[\bar{n}_{\varphi_t}]) = 0$. Then, by Lemma 2, the following holds:

$$\# \Lambda(T_{\varphi_t}[\bar{n}_{\varphi_t}], r_{\varphi_t}) = 4N_a(T_{\varphi_t}[\bar{n}_{\varphi_t}]) + 2\varphi_t + 2r_{\varphi_t} + 12.$$

(2)
Therefore, we obtain the following inequality:

\[ \frac{24}{20} = \frac{3}{2} \geq 2(\varphi_t + r_{\varphi_t} - 2\varphi_u). \]  

The right-hand side of inequality (4) is larger than 0 if \( \varphi_t \geq 2\varphi_u \). Since \( \varphi_t > \sqrt{n} \), \#\Lambda(T_\varphi[n], r) \) is not minimized when \( \varphi > 2\sqrt{n} \).

**Lemma 5.** Let \( n \) and \( \varphi \) be natural numbers such that \( \varphi < n \), and let \( r = n \mod \varphi \) and \( \bar{n} = n - r \). Then, the inverse of Lemma 4 holds, that is, \( T_\varphi[n] + r \) such that \#C(n) > \#\Lambda(T_\varphi[n], r) \) exists when \( n > 8 \).

**Proof.** We show that \( T^*[n] \) such that \#C(n) > \#\Lambda(T^*[n], r) \) exists for any \( n > 8 \). Let \( \varphi_u \) be the maximum integer such that \( \varphi_u \leq \sqrt{n} \), and let \( \bar{\varphi}_u = n - r_{\varphi_u} \). By Lemma 3 and Lemma 4

\[ \#\Lambda(T_\varphi[n], r) \leq \#\Lambda(T_\varphi[\bar{n}], r_{\varphi_u}) \leq 4\varphi_u + 12 \leq 8\sqrt{n} + 12 \]  

If \( n > 24 \), \#\Lambda(T^*[n], r) < \#C(n) \) follows \#C(n) = 2n + 3 > 8\sqrt{n} + 14 and inequality (5). By Table 1, it also holds in the case of \( 8 < n \leq 24 \).

Lemma 5 implies that if \( \varphi^* > 8 \), we can make \( \Lambda(T^*[n], r) \) into a more compact \( \lambda \)-term by applying RTP to \( \varphi^* \) and by translating its result into a \( \lambda \)-term via Algorithm 1. This operation can be applied recursively while \( \varphi^* \), at each recursion step, is larger than 8. By \( \Lambda^*(n) \) we denote the final \( \lambda \)-term obtained as a result. That is,

\[
(\lambda p f x.M_i)C(\varphi^*_{i+1}) := \Lambda(T^*[\varphi^*_i], r_{\varphi^*_i}) \\
\Lambda^*(n) := (\lambda p f x.M_i)((\lambda p f x.M_i) \cdots ((\lambda p f x.M_N)C(\varphi^*_N)) \cdots),
\]

where \( 0 \leq i \leq N \), \( \varphi^*_0 = n \), \( r_{\varphi^*_i} = \varphi^*_i \mod \varphi^*_{i+1} \), and \( \varphi^*_i = \varphi^*_i - r_{\varphi^*_i} \). We show an algorithm generating \( \Lambda^*(n) \) for given \( C(n) \) in Algorithm 3.
Therefore, \( \#\Lambda( \overline{\text{contradiction}} ) \) is wrong and the proposition has been proven.

The function part of \( \Lambda(\overline{\text{and}}) \) and \( \#p \) time, less than or equal to that of \( \Lambda(\overline{\text{the size of Church numerals appearing in the argument parts}}, \#) \) holds.

### Algorithm 2

**Algorithm generating \( \Lambda^*(n) \) for \( C(n) \)**

**Input:** \( C(n) \)

**Output:** \( \Lambda^*(n) \)

```plaintext
1: function COMPACTION(\( C(n) \))
2:    \( n \leftarrow \) the natural number corresponding \( C(n) \)
3: for \( \varphi = 2 \) to \( 2\sqrt{n} \) do
4:    \( r_\varphi \leftarrow n \mod \varphi \)
5:    \( \overline{n} \leftarrow n - r_\varphi \)
6:    \( T_\varphi[\overline{n}] \leftarrow \text{RTP of } \overline{n} \text{ by } \varphi \)
7:    \( \Lambda(T_\varphi[\overline{n}], r_\varphi) \leftarrow \lambda\text{-term translated from } T_\varphi[\overline{n}] + r_\varphi \text{ via Algorithm 1} \)
8: end for
9: \( T^*[\overline{n}] \leftarrow T_\varphi[\overline{n}] \) in minimum \( \#\Lambda(T_\varphi[\overline{n}], r_\varphi) \)
10: \( r^* \leftarrow r_\varphi \) in minimum \( \#\Lambda(T_\varphi[\overline{n}], r_\varphi) \)
11: if \( r^* \leq 8 \) then return \( \Lambda(T^*[\overline{n}], r^*) \)
12: else return \( (\Lambda(T^*[\overline{n}], r^*))(\text{COMPACTION}(\overline{C(r^*)})) \)
13: end if
14: end function
15: main
16: output COMPACTION(\( C(n) \))
17: end main
```

### 3.5 The upper bound of the size

**Lemma 6.** Let \( n \) and \( \varphi \) be natural numbers such that \( \varphi < n \), and let \( r = n \mod \varphi \) and \( \overline{n} = n - r \). Then, the size of the function part of \( \Lambda(T^*[\overline{n}], r) \) is less than or equal to that of \( \Lambda(T_2[\overline{n}], r) \).

**Proof.** We show it by reduction to contradiction. We assume that the size of the function part of \( \Lambda(T^*[\overline{n}], r) \) is larger than that of \( \Lambda(T_2[\overline{n}], r) \). With regard to the size of Church numerals appearing in the argument parts, \( \#C(2) \leq \#C(\varphi^*) \) holds. \( \#\Lambda(T_\varphi[\overline{n}], r_\varphi) \) is the sum of the sizes of the function and argument part. Therefore, \( \#\Lambda(T^*[\overline{n}], r) > \#\Lambda(T_2[\overline{n}], r) \) follows the assumption. However, this contradicts \( \Lambda(T^*[\overline{n}], r) \) being the minimum \( \lambda\)-term of \( \Lambda(T_\varphi[\overline{n}], r_\varphi) \). Hence, the assumption is wrong and the proposition has been proven.

**Theorem 1.** \( \mathcal{O}(\#\Lambda^*(n)) \) is \( \mathcal{O}((\log_2 n)^{\log n/\log \log n}) \) with a natural number \( n \).

**Proof.** Let \( \varphi \) be a natural number such that \( \varphi < n \), and let \( r = n \mod \varphi \) and \( \overline{n} = n - r \). Then, \( \mathcal{O}(\#\Lambda(T^*[\overline{n}], r)) \) is sum of \( \mathcal{O}(\text{the size of the function part}) \) and \( \mathcal{O}(\text{the size of the argument part}) \). First, we consider \( \mathcal{O}(\text{the size of the function part}) \). By Lemma 6, it is bounded by \( \mathcal{O}(\text{the size of the function part of } \#\Lambda(T_2[\overline{n}], r)) \). Here, \( T_2[\overline{n}] \) is as follows:

\[
T_2[\overline{n}] = k^2 T_2[\overline{p}_k] + (k^2) + k^{-1} T_2[\overline{p}_{k-1}] + (k^{-1} 2) + \cdots + 1 T_2[\overline{p}_1] + (1)
\]

where \( \overline{p}_i \) is \( p_i - (p_i \mod 2) \), which is a coefficient of \( \varphi^* \) for \( 1 \leq i \leq k \). At that time, \( p_i \mod 2 \) is 1, at most. Then, the following inequality holds with regard to \( k \):

\[
(slog_2 n) - 1 < k \leq slog_2 n
\]
This says that $k$ is the maximum integer such that $k \leq \text{slog}_n$. By Lemma 4 in the function part, the size increments by addition, multiplication, and exponentiation are $4$, $2$, and $2$, respectively. Therefore, the maximum size of each term $O(2i \cdot \bar{p}_i + 2^i)$ is $2i + 2 + (\text{the size of } \bar{p}_i) + 4 + 2^i$ where $i \leq k$ and these terms appear $k$ times at most. RTP partitions each $\bar{p}_i$ recursively. Let us denote the number of recursion times by $\rho$. With regard to the upper bound of the size of the function part in $\Lambda(T_2[n], r)$, the following holds:

$$O((2k + 2 + (\text{the size of } \bar{p}_n) + 4 + 2k) \cdot k) = O(k^2 + k + (\text{the size of } \bar{p}_n)) = O(k^\rho).$$

Note that $i$ can be $k$ in each recursion step. Therefore, by Lemma 4

$$(k^2)\rho \leq n \iff \rho \log k^2 \leq \log n$$

$$\rho \leq \frac{\log n}{\log^2 2} < \frac{\log n}{\log (\text{slog}_2 n)^{\rho}} = \frac{\log n}{\log \log (\text{slog}_2 n)^2} = \frac{\log n}{\log \log \log n} \quad (7)$$

Hence, the conclusion below follows inequality (4) and (7):

$$O(k^\rho) = O((\text{slog}_2 n)^{\log n / \log \log n}).$$

Secondly, we consider $O(\text{the size of the argument part})$. If $\varphi^* \leq 8$, it is constant because $O(\#C(\varphi^*)) = O(\#C(8))$. If $\varphi^* > 8$, $C(\varphi^*)$ is recursively compacted. By the proof above, the upper bound of the size of the $\lambda$-term result is $O((\text{slog}_2 \varphi^*) \log \varphi^*/\log \log \varphi^* + \#C(\varphi^1))$ where $\varphi^1$ is $\varphi_1$ in minimum $\Lambda(T_2, [\varphi^*], (\varphi^* - (\varphi^* \mod \varphi_1)))$. Here, $\varphi^*$ is clearly less than $n$. This is followed by

$$O((\text{slog}_2 \varphi^*) \log \varphi^*/\log \log \varphi^*) \leq O((\text{slog}_2 n)^{\log n / \log \log n}).$$

If $C(\varphi^1) > 8$, it is also converted recursively. However, a similar inequality holds in each recursion step and the final $\varphi^*$ results in a constant such that $\varphi^* \leq 8$. Therefore, $O(\#\Lambda^*(n))$ is $O((\text{slog}_2 n)^{\log n / \log \log n})$.

4 Application to Higher-Order Compression

4.1 Abstract of higher-order compression

In higher-order compression, first, the source text data is regarded as the $\lambda$-term such that terminal alphabets are repeatedly combined form tail by functional applications. For example, the source text $\text{ababc}$ is regarded as the $\lambda$-term $\lambda \lambda (\lambda b (a (b (c \$))))).$. Then, we convert it to another $\lambda$-term whose size is less than the size of the original, with the remaining equivalency between them. Here, we say that there is an equivalency between $M$ and $N$ if both $M$ and $N$ result in the same $\lambda$-term by calculation.

4.2 Relationship between Church numerals and repetition patterns

In the $\lambda$-terms of higher-order compression, repetition patterns appear as $((\text{a Church numeral corresponding to the repetition number of the pattern}) (\text{pattern})).$ For example, the $\lambda$-term corresponding to the string "$\text{ababcabc}$" is expressed as follows:

$$((\text{a b c})(\text{a b c})(\text{a b c})(\text{a b c}(\text{a b c} \$))) \rightarrow (\lambda f.x.f(f(f(f x))))(\text{a b c} \$) = C(4) (\text{a b c} \$)$$

10
If the repetition number is large, i.e., the Church numeral appearing in the \(\lambda\)-term is large, we can then compress the \(\lambda\)-term by compacting the Church numeral via our algorithm.

### 4.3 A method using binary expression

As related work, we introduce a method that can compress Church numerals using a binary expression. This method is reported by Kobayashi et al. [1]. Let 
\[
\begin{align*}
    b_0 &= (\lambda f pq. f \, p \, p \, q) \quad \text{and} \\
    b_1 &= (\lambda f pq. p \, (f \, p \, q)).
\end{align*}
\]
Then, the term 
\[
(\underbrace{b_1(b_0(b_1(b_1(b_1(C(0))))))}_{57}) \quad \text{generates the} \quad \lambda\text{-term, which expresses} \quad a^{57},
\]
that is, 
\[
(\underbrace{a \,(a \cdots (a \, \cdots))}_{57}).
\]
Here, the binary expression of 57 is 111001, and the part 
\[
(\underbrace{b_1(b_0(b_1(b_1(b_1(C(0))))))}_{57})
\]
corresponds to its reverse.

Ahead of Kobayashi et al., Mogensen [2] proposed a binary expression of Church numerals and generalized to higher number-bases. The above method is essentially the same as the method of Mogensen.

### 4.4 Experiment

To confirm the efficiency of our Algorithm, we conduct an experiment with regard to the compression ratio. Figure 1 depicts the result of the experiment.

We used the artificial data \(a^n\$\) as input. In Fig. 1 the horizontal axis shows the repetition \(n\) and the vertical axis shows the size of each \(\lambda\)-term. "string" denotes 
\[
\#(a \,(a \cdots (a \, \cdots)));
\]
"binary" denotes the size of the \(\lambda\)-term compacted by the method using the binary expression. "YKS" denotes the compaction manner of higher-order compression, proposed by Yaguchi et al. [4], and, "proposed" denotes our proposed method that uses Algorithm 2.

The inequality (proposed \(\leq\) binary) holds in 9625 cases out of 10000 cases. The ratio (proposed / binary) is approximately 0.7887. This means that the size of the compressed \(\lambda\)-term decreased by approximately 21%, on an average, by our algorithm. Similarly, the inequality (proposed \(\leq\) YKS) holds in 9584 cases and the ratio (proposed / YKS) is approximately 0.7667. This means that the size decreased by approximately 23%, on an average, by the proposed method.
5 Conclusion

In this paper, we addressed the problem of compacting Church numerals, which is useful for higher-order compression. We proposed RTP for decomposing large numerals and presented a λ-term conversion algorithm using RTP. The λ-terms produced by the algorithm tend to become shorter in practice, while its theoretical size, $O(s\log_2 n \log n / \log \log n)$, may grow larger than $O(\log n)$ in the worst case. For $n = k2$, of course, $C(n)$ is converted to the λ-term whose size is $O(s\log_2 n) = O(k)$. We have not proved the lower bound of the size in the worst case; it is one of our future works.

Very recently, Takeda et al. [3] proposed an efficient method to encode λ-terms. Efficiently finding repeating regions in an input and counting the number of repetitions are the problems remaining in higher-order compression.

Acknowledgements. The authors would like to thank Ayumi Shinohara and his colleagues for providing the source codes for higher-order compression. This work was supported by JSPS KAKENHI Grant Number JP15K00002 and JST CREST Grant Number JPMJCR1402, Japan. Finally, the authors would like to thank Enago (www.enago.jp) for the English language review.

References

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

[2] Torben Æ. Mogensen. An investigation of compact and efficient number representations in the pure lambda calculus. In Revised Papers from the 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics: Akademgorodok, Novosibirsk, Russia, PSI ’02, pages 205–213, London, UK, 2001. Springer-Verlag.

[3] Kotaro Takeda, Naoki Kobayashi, Kazuya Yaguchi, and Ayumi Shinohara. Compact bit encoding schemes for simply-typed lambda-terms. SIGPLAN Not. (Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming), 51(9):146–157, September 2016.

[4] Kazuya Yaguchi, Naoki Kobayashi, and Ayumi Shinohara. Efficient algorithm and coding for higher-order compression. In In proceedings of 2014 Data Compression Conference (DCC2014), pages 434–434, March 2014.
This figure "figure_1.png" is available in "png" format from:

http://arxiv.org/ps/1706.10061v1