二進表現と二分木の間の効率的でクリーンな可逆変換

大久保 雄飛　横山 哲郎　金山 知俊

可逆計算システムで用いる可逆アルゴリズムのひとつとして，二分木の列挙やランダム生成に有用な二分木のランク・アンランク計算のクリーン可逆シミュレーションを提案する．二分木のランク計算やその逆計算を行う非可逆アルゴリズムは 1970 年代から研究が行われてきており，これらの成果を用いてランク計算の関数とその逆関数のそれぞれの可逆シミュレーションを組み合わせてクリーン可逆シミュレーションを行うことはできる．しかし，こうした可逆シミュレーションでは，複数回のパスおよびに実行時間に比例した大きさのメモリが必要である．二分木のランク・アンランク計算は 1977 年に Knott によって報告されたアルゴリズムによるものが最初であるとされているが，そのクリーン可逆シミュレーションは 1 パスで漸近的な実行時間とメモリ使用量が元のアルゴリズムと同じものがすでに知られている．われわれは 1985 年に Er によって報告された，二分木のランク・アンランク計算のアルゴリズムに対して，1 パスで漸近的な実行時間とメモリ使用量が変わらないクリーン可逆シミュレーションを提案する．本稿中の可逆プログラムは，可逆言語 Janus の処理系で動作を確認できる．

【要修正】We propose a clean reversible simulation of ranking and unranking binary trees as a reversible algorithm for reversible computing systems, which is useful for enumerating and randomly generating binary trees. Algorithms for ranking binary trees and their inverses have been studied since the 1970s, and the results have been combined to realize a clean reversible simulation. However, such a reversible simulation requires multiple paths for traversing the given data and/or intermediate data as well as additional storage proportional to the length of the computation. The proposed reversible simulation involves only one path, and its asymptotic running time and memory usage are equivalent to those of the original. All the reversible programs presented in this paper can be run on an interpreter of the reversible programming language Janus.

1　はじめに

可逆計算システムでは，可逆演算を基本とするとその上で使うアルゴリズムの設計が容易になる．可逆性制約が考慮された可逆アルゴリズムの設計と解析には，その制約があるので通常のアルゴリズムの設計および解析とは異なる手法が必要である．一般的に，(非可逆）アルゴリズムを可逆計算システム上で可逆シミュレーションするような互いに時間計算量・空間計算量などがトレードオフ関係にある一般解法が知られている [3] [8] [25] [26]．しかし，こうして得られる可逆シミュレーションでは，計算中に必要なメモリおよび元のアルゴリズムでは出力されないゴミ出力の量が実行時間に比例した大きさにまでなってしまう問題がある．一方，問題固有の情報を考慮して可逆アルゴリズムを設計するとこうしたものを小さくできる．任意の単射関数に対して，必ずゴミ出力無しの，すなわちクリーンな可逆シミュレーションが作れることが知られている．しかし，任意の非可逆プログラムを同等の計算複雑度でゴミ出力が最適な可逆プログラムにする体系的な方法は知られていない．したがって，少なくとも現在のところ，クリーン可逆シミュレーションの効率化は，個別のアルゴリズムについて経験や勘を頼りに手動で行わざるを得ない．

木構造は節点の間の「枝分かれ」によって情報を配

* Efficient Clean Reversible Transformation between Bit Patterns and Binary Trees
This is an unrefereed paper. Copyrights belong to the Authors.

Yuhi Ohkubo, Tetsuo Yokoyama, Chishun Kanayama, 南山大学理工学部ソフトウェア工学科, Department of Software Engineering, Faculty of Science and Engineering, Nanzan University.

置する構造であり，計算システムやアルゴリズムにおいて最も重要な構造のひとつである．たとえば，代数式の表現やアルゴリズムの解析に木構造を用いる手法が知られている [19]．本稿では，二分木のランク・アンランク計算のクリーンな可逆アルゴリズムを設計・解析する．二分木のランク・アンランク計算は，一様分布したランクに対応する二分木をランダムに生成することに応用できたり二分木の列挙のアルゴリズムの基礎になったりするアルゴリズムである [28] [18] [19]．このように応用範囲が広いにもかかわらず，筆者の知る範囲では，その可逆アルゴリズムの提案はこれまでほとんどなされてこなかった．

文献 [28] によると，二分木のランク・アンランク計算は，Knott によるアルゴリズム [18] が最初である．このアルゴリズムに対する，1 パスで漸近的な実行時間とメモリ使用量が同じであるようなクリーン可逆シミュレーションが提案されている ?[]．この手法が他の二分木のランクに対するランク・アンランク計算の可逆化に直接的に役立つかはよく分かっていない．

本稿では，Er によるランク計算とアンランク計算の Pascal プログラムの両方の可逆シミュレーションを考える．これらを融合することで効率的な可逆アルゴリズムを得る．具体的には，文献 ?[] の導出法に従って，非可逆アルゴリズムから埋め込みによって可逆シミュレーションを得て，次に Bennett 法でそれらを組み合わせることでクリーン可逆シミュレーションを得る．さらに，漸近的な実行時間と使用メモリ量が元のアルゴリズムと同じという意味で効率的な可逆アルゴリズムを提案する．

本稿のすべての Janus プログラムの完全版は，われわれのウェブサイト `http://tetsuo.jp/ref/XXXX2017/` のリンク先で閲覧することができる．また，そのウェブサイトのリンク先に設置された Janus のオンラインインタプリタで実行をして動作を確認することができる．

2 関連研究

可逆計算の研究が始まった動機のひとつは，計算に必要なエネルギー量や発熱量の特定と削減であった [11] [21] [4]．非可逆計算における情報の消去には（正の）発熱が伴うというランダウアの原理 [20] が観測されている [5]．一方，可逆計算は情報の消去を原理的に回避できることが 40 年以上前から知られている [3]．もちろん，現在の電子回路では論理演算以外に多くのエネルギーを費やしているが，論理的状態を微視的状態で実現できると可逆性は重要な特性になる．可逆計算は，量子計算の特別な場合であり，量子計算や量子回路のテキストでも解説される [16] [8]．

物理的な可逆性を論理レベルで扱うには，論理回路 [27]，計算機アーキテクチャ [29]，プログラミング言語（[22] [15] [2] [13] [17] [33] など），およびアルゴリズムのすべての階層を可逆に設計する必要がある．しかし，可逆アルゴリズムの研究は次のような一部の例外を除くとこれまで深くは行われてこなかった．

非可逆アルゴリズムで失われるデータや制御情報をすべて憶えるようにしたアルゴリズムは可逆になる [3]．この変換を（非可逆計算の可逆計算への）埋め込みと呼ぶ．埋め込みによる可逆シミュレーションの実行には，時間計算量，空間計算量，消去情報量，および実時間性などの観点でトレードオフをもつ手法が提案されている（たとえば，文献 [13] [26] およびその参考文献を参照）．

埋め込みよりも効率的で特定のアルゴリズム族に適用可能な可逆シミュレーションが存在する．任意の入力 x に対する関数 f の返却値 y とその返却値 y を入力としたその逆関数 f^{-1} の返却値として x が一意に得られる場合，両者の可逆シミュレーションのゴミ出力が必ず一致するならば，2 パスの可逆シミュレーションを構成できる [32]．たとえば，この可逆シミュレーションを利用して，2 パスの可逆な range coding が実現された．また，任意の比較ソートには，ゴミ出力量が最小で 2 パスの可逆シミュレーションが存在する [1]．可逆論理回路の設計では，特定の関数族に適用が可能な可逆シミュレーションが提案されている（たとえば，[30] を参照）．たとえば，Mealy 機械を用いた手法で in-place に定数倍をする乗算器を生成する方法がある [23]．

これらの一般解法よりも問題固有の情報を使うことで効率的な可逆アルゴリズムを実現できる．挿入整列法，バブルソート，選択整列法，併合整列法，ク

イックソートといった比較ソートは複数の整列前の入力列が同一の整列された出力列となるので非単射であるので，それらの可逆シミュレーションにはゴミ出力が必要であるが，最悪ゴミ出力量が最小になる可逆アルゴリズムが知られている [22] [9] [25] [1].

可逆計算における元々の問題とは無関係に計算の可逆性は計算機科学において便利な概念である．可逆計算モデルには，多くの非可逆な計算モデルにそれぞれ対応するものが知られている（たとえば，文献 [35] とその参考文献を参照）．他にも，可逆性は，計算複雑性理論，可逆デバッグ †1，プログラム逆変換 [15] [14] [24]，データベースのビュー更新問題や双方向モデル変換 [12] で有用な性質である．

3 可逆プログラミング言語 Janus

本稿では，命令型可逆プログラミング言語 Janus の拡張版 [31] でプログラムを書いてある（以下，単に Janus と呼ぶことにする）．Janus は，C 言語に似た構文に加えて可逆性を保証するための構文要素をもつ．本稿で用いる Janus は，非可逆なプログラムが記述できない（単射関数しか表せない）という意味で可逆であることが証明されている．

例を通して可逆プログラミングを一瞥する．$n\ (\geq 0)$ 番目のカタラン数 $B_n = \frac{1}{n+1}\binom{2n}{n}$ は漸化式

$$B_n = 4B_{n-1} - \frac{6B_{n-1}}{n+1} \qquad (n \geq 1) \qquad (1)$$

によって効率的に計算できることが知られている（文献 [18] および文献 [19] の 2.3.4.4 節を参照）．以下の可逆プロシージャは，正整数 n とゼロクリアされた配列 B を受け取り †2，カタラン数列 $B_0, B_1, \ldots, B_{n-1}$ を B[0:n-1] に格納する:

```
1 procedure mk_catalan_tbl(int B[],int n)
2   B[0] ^= 1
3   local int i = 0
4     from  i = 0 do
5       i += 1
6     loop
7       B[i] ^= 4*B[i-1] - 6*B[i-1]/(i+1)
8     until i = n
9   delocal int i = n
```

†1 たとえば，https://www.gnu.org/software/gdb/news/reversible.html, http://caml.inria.fr/pub/docs/manual-ocaml/debugger.html

†2 ゼロクリアとは，変数の値が 0 であったり，配列の全要素の値が 0 であることを意味する．

Janus の構文と意味論の概要を示す．変数の基本型は，整数型，整数型の配列とスタックである．Pascal で使われるような単純代入 `x := 1` は使えず，2, 5, 7 行目のように C 言語でも使われる複合代入演算子 `+=`, `-=`, `^=` を使わねばならない．ただし，複合代入演算子式 $x \oplus$= e の左オペランド x で指し示されるオブジェクトが右オペランドの式 e に出現してはならず，左オペランドが添え字演算子式の場合 (x[e_i] $\oplus$=e) はその添え字 e_i にも出現してはならない．この制約は `x -= x` のような非単射な計算を行う文を記述できないようにする．プッシュ `push(x,g)` は，スタック `g` の先頭へ変数 `x` の値を追加して `x` の値を 0 にする．ただし，`push(`c`,g)` の c が定数である場合は，その定数の値をスタック `g` の先頭へ追加する．ポップ `pop(x,g)` は，スタック `g` の先頭から取り除いた値をゼロクリアされた変数 `x` に格納する．スタック `g` の先頭要素の値は式 `top(g)` によって求められる．制御文において制御の合流地点では実行時アサーションを使うことで可逆性を保証する．可逆選択文 `if` e_1 `then` s_1 `else` s_2 `fi` e_2 は通常の選択文のように動作するが，アサーション e_2 の値が `then` 節が実行された後には真，`else` 節が実行された後には偽でなければならない．可逆繰り返し文 `from` e_1 `do` s_1 `loop` s_2 `until` e_2 では，アサーション e_1 が，実行前に真，その後は繰り返すごとに偽と評価されなければならない．`do` 節 s_1 は e_1 と e_2 の評価の間に，`loop` 節 s_2 は e_2 と e_1 の評価の間に実行される．`then` 節，`else` 節，`do` 節，および `loop` 節は，空文のみをもつ場合は省略される．`local` 節は，指定した局所変数を指定した式の値で初期化する．`local` 節と対になる `delocal` 節では，局所変数の値と指定した式の値が等しいことが要求され，局所変数がスコープから除かれる．`call` 文は，プロシージャを呼び出す．引数は参照渡しされる．`uncall` 文は，プロシージャの逆呼び出しを行う．すなわち，`call` 文とは逆順にプロシージャ本体のそれぞれの文を逆実行する．プロシージャの逆呼び出しに必要な時間計算量および空間計算量は元のプロシージャと同じである．プロシージャの呼び出し・逆呼び出しの引数渡しには参照渡しが使われる．また，任意の Janus のプロシージャは再帰下降的にプログラム

逆変換によって自身と同じ時間計算量および空間計算量の逆プログラムに変換することが可能である [34].

プロシージャ`mk_catalan_tbl` は本体の中のループが n 回繰り返されるので実行時間が $\mathrm{O}(n)$ である．$B_0,\ldots,B_{n-1}$ が `B[0:`$n-1$`]` に格納されている場合，プロシージャ逆呼び出し `uncall mk_catalan_tbl(B,n)` によって `B[0:`$n-1$`]` はゼロクリアされる.

任意の関数 $f : X \to Y$ に対して，ある G が存在して任意の $x \in X$ に対して $\mathit{fst}(f'(x)) = f(x)$ となる単射関数 $f' : X \to Y \times G$ が存在する．ただし，fst は $\mathit{fst}(x,y) = x$ となる射影関数である．必ずしも単射にならない関数 f から上記の条件を満たす単射関数 f' を作ることを単射化と呼ぶ．G の要素は，元の関数 f の出力には出現しないことからゴミ出力と呼ぶ．特に，G がスタックである場合，G に属するスタックをゴミスタックという.

可逆言語 `R` のプログラム `q` が非可逆言語 `IR` のプログラム `p` の可逆シミュレーションであるとは，任意の x に対して $\mathit{fst}([\![\mathtt{q}]\!]^{\mathtt{R}}(x)) = [\![\mathtt{p}]\!]^{\mathtt{IR}}(x)$ であること，すなわち $[\![\mathtt{q}]\!]^{\mathtt{R}}$ が $[\![\mathtt{p}]\!]^{\mathtt{IR}}$ から単射化された関数であることをいう．可逆言語 `R` の可逆シミュレーション `q` がクリーンであるとは，`q` の出力が元の出力のみでゴミが含まれないことをいう.

任意の非可逆プログラムに対して，制御情報や消去されるデータを計算履歴としてゴミスタックに貯めることによる埋め込みをして，その可逆シミュレーションを得ることができる．以下は，埋め込みをしてその後に一部効率化されたプロシージャで，`call catalanE(b,g)` で呼び出されると，ゴミスタック `g` に不要なデータを貯めながら b 番目のカタラン数を求めてその値を変数 `b` に格納する:

```
1  procedure catalanE(int b,stack g)
2    if b=0 then
3      b ^= 1 // B0
4      push(0,g)
5    else
6      local int n = b
7        b -= 1
8        call catalanE(b,g)
9        local int b1 = b
10         push(b,g)
11         b ^= 4*b1 - 6*b1/(n+1) // 式 1 より
12         push(b1,g)
13       delocal int b1 = 0
14       push(n,g)
15     delocal int n = 0
16     push(1,g)
17   fi top(g)=0
```

17 行目の選択文のアサーション `top(g)=0` は，`then` 節と `else` 節のそれぞれ最後のプッシュ文でゴミスタック `g` に追加された互いに異なる制御情報によって必ず成立する．また，式 1 の漸化式にそのまま対応するような単純代入式の文 `b := 4*b - 6*b/(n+1)` は Janus では記述できないので，9–13 行において局所変数 `b1` の導入とプッシュ文によって不要なデータをゴミスタック `g` に移すことで `b` に格納されたカタラン数を更新している．さらに，何番目のカタラン数を計算しているかを記憶した局所変数 `n` は 15 行目で解放される直前でその不要になった値がゴミスタック `g` に移されている．このように制御情報と不要データを共にゴミスタックに移動させることで，常に埋め込みによる可逆シミュレーションは実現できる．問題は，入力される値 b に対して実行時間 $\mathrm{O}(b)$ が必要であるが，メモリもこれに比例した量だけ使用しなければならないことである．なお，本稿では埋め込み（Embedding）によって作成されたプロシージャを区別しやすくするためにその接尾辞に `E` を用いている.

本稿では以下の記法を用いる．コードは `typewriter font` で，数式・意味領域の値・メタ変数は *italic font* で記す．同じ識別子でも異なるフォントの場合は意味が異なる．たとえば，変数 `i` の値を i と記す．長さ n の配列名 `p` の配列を `p[0:`$n-1$`]` と記す．配列 `p[0:`$n-1$`]` が与えられた場合，$p[i:j]$ は，$i \leq j < n$ の場合は配列 `p[0:`$n-1$`]` の i 番目から j 番目の要素の値の列とし，$i > j$ の場合は空列とする．すべての要素が s である部分配列の値を $(s,\ldots,s)$ と記す.

以降では，表記を簡潔にするために，カタラン数列 $B_0,\ldots,B_{n-1}$ を格納した配列 `B[0:`$n-1$`]` の要素はどこでも読み込めるものとする.

4 二分木とランク計算

本章では，まず本稿で扱う（ラベル無し）二分木の定義を与える．また，二分木を自然順序で列挙したときの列の添え字に対応するランクの計算方法と，その

逆に，ランクに対応する二分木の計算方法を紹介する．さらに，以降の章で用いる二分木の表現である木置換について説明する．これらは基本的に文献[18] に掲載されたものであり [†3]，次章で議論する効率的な可逆シミュレーションのために詳解したものである．

4.1 二分木の定義と二分木の順序と節点数

（ラベル無し）二分木とは次の 1, 2 によって再帰的に定義される節点の有限集合 T である．1) T は空集合である．もしくは，2) 根としての節点および共通部分を含まない 2 つの集合 T_l, T_r の順に一意に分割され，T_l, T_r は二分木である．T_l, T_r をそれぞれ二分木 T の左部分木，右部分木と呼び，$l(T), r(T)$ と表す．二分木 T の節点数を $|T|$ と表す．特に，$|T| = 0$ の二分木 T を空木と呼ぶ．

二分木の上に同値関係や順序を定義する．二分木 T_1, T_2 が一致するとき $T_1 \equiv T_2$ と表す．二分木の二項関係 $T_1 \lessdot T_2$ は以下のように再帰的に定義される:

1. $|T_1| = 0$ かつ $|T_2| > 0$,
2. $|T_1| > 0, |T_2| > 0$ かつ $l(T_1) \lessdot l(T_2)$, もしくは
3. $|T_1| > 0$, $|T_2| > 0$, $l(T_1) \equiv l(T_2)$, かつ $r(T_1) \lessdot r(T_2)$.

$\lessdot$ は木の上の辞書順（もしくは局所順序）と呼ばれる．本稿では，木の順序関係を表すのに辞書順を用いる．

節点数 n の二分木の集合を $\mathcal{T}(n)$ と表す:

$$\mathcal{T}(n) = \{T \mid |T| = n\} \tag{2}$$

$\mathcal{T}(n)$ の元の数，すなわち節点数 n の二分木の数は n 番目のカタラン数 B_n になる．

節点数 n の二分木 T $(\in \mathcal{T}(n))$ は，根以外の全ノードが次数 3 となるように空木を追加することで，完全二分木 $\hat{T}$ になる．任意の n と T $(\in \mathcal{T}(n))$ に対して $\hat{T}$ は $2n$ ビットのビットパターン $A_n = \langle a_1, a_2, \ldots, a_{2n}\rangle$ で一意に表現できる．前順走査で節を 1, 葉を 0 とそれぞれ表現するのである．ただし，最後に走査するのは必ず葉であり，その分の 0 は表現に加えないことにする．二分木からビットパターンへの

†3 文献[18] では木置換の要素とランクは共に 1 から始まるが，本稿では簡単な記述を目的に 0 から始める．

4.2 二分木のランク

二分木 T のランク $rank'(T)$ とは，節点数が $|T|$ であり T よりも自然順序で小さい二分木の数である [†4]:

$$rank'(T) = \#\{S \mid S \prec T, S \in \mathcal{T}(|T|)\} \tag{3}$$

すなわち，ある一定の大きさのすべての二分木を自然順序で並べた列を考えた場合，それぞれの二分木に対してその列の先頭要素を 0 番目とする添え字がそれぞれの二分木のランクである．二分木 T に対する $rank'(T)$ は，節点数 $|T|$ の二分木において T に対応する $B_{|T|} - 1$ 以下の一意の 0 以上の整数である．たとえば，図 1 には節数 4 の二分木が自然順序順，すなわちランクの昇順に網羅的に列挙されている．ランク 3 とランク 4 の二分木は，条件 3, 2, 2, 1 の順に使用することで自然順序順であることを確認できる．

$rank'(T)$ は再帰関数

$$\begin{aligned} rank'(T) = \mathbf{if}\ |T| = 0\ \mathbf{then}\ 0\ \mathbf{else} \\ \Big(\sum_{0 \le j < |l(T)|} G_{j,|T|} \\ + rank'(l(T)) \times B_{|r(T)|} \\ + rank'(r(T))\Big) \end{aligned} \tag{4}$$

によって計算できる．**then** 節の項が 0 であることは，空木より小さい木がないことに対応する．**else** 節の 3 つの項は，それぞれ

(a) $|l(T')| < |l(T)|$

(b) $|l(T')| = |l(T)|$ かつ $l(T') \prec l(T)$

(c) $l(T') \equiv l(T)$ かつ $r(T') \prec r(T)$

という条件の二分木 T' の数に対応する．条件 (a) を満たす二分木の左部分木の節点数は $j = 0, 1, \ldots, l(T) - 1$ の範囲になる．第 1 項は，そうした二分木の数 $G_{j,|T|}$ の和である．第 2 項は，$l(T') \prec l(T)$ を満たす左部分木 $l(T')$ の数 $rank'(l(T))$ とそうした各左部分木に対応する右部分木の数 $B_{|r(T)|}$ の積である．第 3 項は，$r(T') \prec r(T)$ を満たす右部分木 $r(T')$ の数 $rank'(r(T))$ である．条件 (a), (b), (c) は互いに排反であり，これらの条件を満たす二分木の数の和は，T と同じ節点数で，T より自然順序で小さい二分木の数である $rank'(T)$ に一致する．

たとえば，図 2 の二分木 T のランクは，左部分木のランクが 0 であり，図 1 より右部分木のランクが 7

†4 $\#X$ は集合 X の要素数を表す．

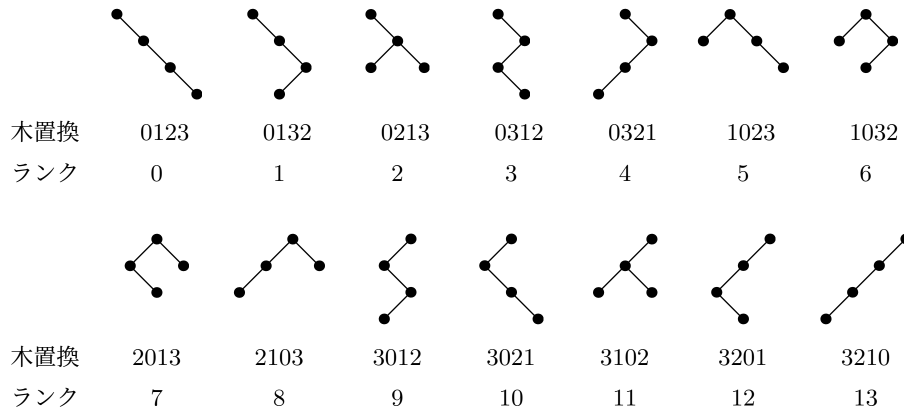

図 1　自然順序順に並べられた節数 4 の二分木

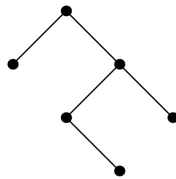

図 2　ランク 49 の二分木（木置換 104235 に対応する）

であることから，49 ($= G_{0,6} + 0 \times B_4 + 7$) である．（節に付属するラベルは 4.4 節で用いる.）

特に，二分木 T が同じ二分木を子どもにもったとしても，$l(T)$ が空集合である場合と $r(T)$ が空集合である場合，すなわち左部分木のみをもつ場合と右部分木のみをもつ場合ではそれぞれ別の二分木であり，それらのランクが互いに異なることに注意して欲しい．

異なる節点数の二分木が同じランクをもつことはあるが，節点数が与えられた場合はそのようなことはなく，任意の n について $rank' : \mathcal{T}(n) \to \{0, 1, \ldots, B_n - 1\}$ は単射である．したがって，与えられた二分木の節点数とランクを組で返すようにした関数

$$rank(T) = (|T|, rank'(T)) \qquad (5)$$

は単射である．次章では $rank$ の可逆シミュレーションを考える．

4.3　ランクから二分木の計算

$rank$ は単射なのでその逆関数が存在する．$rank(T)$ の逆関数 $unrank(n, r)$ は，節点数 $n = |T|$ と T のランク r が与えられたら式 4 の $rank'(T)$ の再帰的定義を用いて以下のように T を構成する方法で計算できる．

もし $n = 0$ かつ $r = 0$ であるならば，これらは **then** 節で計算されたものであり，T として空木を構成して終了する．

else 節で計算されたものについて考える．任意の n (≥ 1) に対して $0 \leq j \leq n-1$ なら $G_{jn} > 0$ であり，$0 \leq rank'(l(T))$ および $0 \leq rank'(r(T)) < B_{|r(T)|}$ であることから，第 2 項と第 3 項の和を $B_{|r(T)|}$ で割ると商が $rank'(l(T))$，剰余が $rank'(r(T))$ と一意に決まる．また，第 2 項と第 3 項の和の最大値 $(B_{|l(T)|} - 1) \times B_{|r(T)|} + (B_{|r(T)|} - 1)$ は $G_{|l(T)|,|T|} = B_{|l(T)|} \times B_{|r(T)|}$ より 1 小さい．したがって，第 1 項から第 3 項までの和から $G_{j,|T|}$ を $j = 0, 1, \ldots$ の順で繰り返し減算を行っていって初めて $G_{j,|T|}$ 未満になった場合の j が $|l(T)|$ である．$|r(T)| = |T| - |l(T)| - 1$ より $|r(T)|$ が求まる．$|l(T)|$ と $|r(T)|$ の情報を用いてそれぞれ $rank'(l(T))$ と $rank'(r(T))$ について以上を再帰的に計算することで得られる $l(T)$ と $r(T)$ から T を構成できる．

$rank'(T)$ の **then** 節と **else** 節の値は排反ではないことに注意して欲しい．ランクが 0 でも節点数が 1 以上の場合（$r = 0$ かつ $n \geq 1$ の場合）は **else** 節によって二分木を構成しなければならない．

4.4　木置換による二分木の表現

本稿では（ラベル無し）二分木を木置換で表現する．木置換 $p[0 : n-1]$ は，集合 $\{0, 1, \ldots, n-1\}$ の

置換であり，以下のように構成される：

(i) 0 個の節からなる空木を表す空集合（$n = 0$）の要素からなる空の順序組である．

(ii) もしくは，3 つの列が連接された sp_lp_g である．ここで，s は自然数であり，p_l は s 未満の要素からなる木置換であり，p_g は s より大きい要素からなり各要素から s を減じると木置換である．

2 番目の構成方法において，s が根に，p_l が左部分木に，p_g の各要素から s を減じたものが右部分木にそれぞれ対応する．

たとえば，図 2 に示されたランク 49 の二分木に対応する木置換は 104235 である．この木置換において，1 が根に 0 が左部分木に 4235 の各要素から 1 を減じた 3124 が右部分木に対応する．

本稿では，二分木とその木置換による表現を同一視する．次章以降のプログラムでは，節点数 n の（ラベル無し）二分木を配列 p[0:$n-1$] に格納された木置換で表す．特に，空木を表す空の順序組は $1 \leq a \leq n$ を満たす a を用いて空列 $p[a : a-1]$ で表す．

文献 [18] のランクおよびアンランクを計算する Algol プログラムは入出力される変数を読み替えると次の条件を満たすプロシージャ rankA および unrankA と見なせる．節点数 n の二分木 T に対応する木置換が $p[0 : n-1]$ の場合を考える．この場合，

$$[\![\texttt{rankA}]\!]^{\texttt{Algol}} : (p[0 : n-1], n) \mapsto (n, r) \quad (6)$$

となり，$r = rank'(T)$ である．また，節点数 n のランク r に対して

$$[\![\texttt{unrankA}]\!]^{\texttt{Algol}} : (n, r) \mapsto (p[0 : n-1], n) \quad (7)$$

となり，二分木 $T = unrank(n, r)$ に対応する木置換が $p[0 : n-1]$ である．次章では rankA と unrankA の可逆シミュレーションを考える．

5 二分木の可逆なランク計算

二分木のランク計算とその逆計算の方法を前章でみた．本章では，まず，Bennett 法[3] にしたがって，それら 2 つの計算の埋め込みによる可逆シミュレーションと，それらの可逆シミュレーションの組み合わせによる複数パスのクリーン可逆シミュレーションを実現する．次に，われわれは 1 パスで使用メモリが少ない可逆シミュレーションを提案する．

```
1 //木置換p[a:a+n-1]-(s,...,s)のランクをrに
2 procedure rE(int p[],int n,int a,int s,
      int r,stack g)
3  local int j = 0
4    // (a) 符号化した|l(T)|をrに格納
5    from j = 0 loop
6      r += B[j]*B[n-j-1] // G_jnをrに追加
7      j += 1
8    until j = p[a]-s      // |l(T)|
9
10   // (b) rank'(l(T))の情報をrに追加
11   if j >= 1 then       // |l(T)| ≥ 1の場合
12     local int r1 = 0
13       call rE(p,j,a+1,s,r1,g)
14       r += r1 * B[n-j-1]
15       push(r1,g)
16     delocal int r1 = 0
17   fi j >= 1
18
19   // (c) rank'(r(T))をrに追加
20   if n-j-1 >= 1 then    // |r(T)| ≥ 1の場合
21     call rE(p,n-j-1,a+j+1,s+j+1,r,g)
22   fi n-j-1 >= 1
23  delocal int j = p[a]-s
24
25 // 木置換p[0:n-1]のランクをrに
26 procedure rankE(int p[],int n,int r,
      stack g)
27  call rE(p,n,0,0,r,g)
28  call clear_array(p,n,g) //配列pをクリア
```

図 3 ゴミスタックを使った木置換のランク計算

5.1 埋め込みによるランク計算

図 3 の 26–28 行目に示された rankE は，rankA の計算履歴を記憶することで Janus の可逆プロシージャで実現したものである．この可逆プロシージャでの計算履歴は，具体的には，使わなくなった局所変数と入力された木置換の配列に格納されたデータである．プロシージャ呼び出し rankE(p,n,r,g) における引数は，それぞれ木置換を格納した配列 p, 木置換の長さ n, ランクの計算結果を格納するための変数 r, 計算途中のゴミ情報を格納するためのゴミスタック g である．

図 3 の 2–23 行目に示された補助プロシージャ rE(p,n,a,s,r,g) は部分配列の値 $p[a : a+n-1]$ の各要素からオフセット s を引いた木置換のランクを計算する．プロシージャ rE の呼び出しによってランクが変数 r に格納されたのちに，元の出力では必要のないデータである配列 p[0:$n-1$] のすべての要素を 28 行目のプロシージャ呼び出し clear_array(p,n,g)

によってゴミスタック g に移す．

ランクの計算は，補助プロシージャ rE で行われる．プロシージャ rE は局所変数 j へのメモリの割り当て，上述の条件 (a), (b), および (c) に対応する 3 つのコード片，ならびに局所変数 j のメモリの解放からなる．ここで，木置換 $p[a : a+n-1] - (s, \ldots, s)$ に対応する二分木を T とする．局所変数 r1 はランクを計算するための一時的な値を，局所変数 j は T の左部分木の節点数 $|l(T)|$ をそれぞれ記憶するのに用いられる．ここで，8 行目の式 p[a]-s の値は，木置換 $p[a : a+n-1] - (s, \ldots, s)$ の先頭要素であり，T の左部分木の節点数を値としてもつ．

図 3 のプログラムは，単純な埋め込みを行っただけではなく，制御の流れを変えることなく行える効率化を行ってある．たとえば，6 行目では，元のプログラムにおいて代入によって行われていた値の上書きを複合代入演算子 += を用いて実現している．また，ランクを一時的に記憶するための局所変数は 1 箇所のみに使用した．さらに，選択文や繰り返し文のアサーションは，それぞれの場所で使用できる変数や配列を用いて勘と経験を頼りにして構成することができた．次節のプログラムでもこうした効率化を断り無く行っている．

これらの可逆プロシージャは，元の Algol プロシージャと比較すると，制御の流れが同じであるために，漸近的な実行時間は任意の入力に対して一致する．しかし，15 行目では，以降では使われないデータがゴミスタック g にプッシュされている．このために実行時間に比例したメモリ使用量が追加で必要になってしまっている．

実行時間を悪化させることでメモリ使用量を削減する効率化は可能ではある．15 行目の push(r1,g) をその 2 行前のプロシージャ呼び出しに対応するプロシージャ逆呼び出しに置き換えても元の出力を計算する可逆シミュレーションが実現できる．この実現は，call-copy-uncall の組み合わせからなる局所的 Bennett 法[34] を利用しているといえる．プロシージャ呼び出しによって計算された r1 とゴミスタックへ追加されたデータが，同一のプロシージャの逆呼び出しによって可逆的に消去されるからである．しかし，再帰的にプロシージャが呼び出されるたびに，同一プロシージャが呼び出しと逆呼び出しの 2 回使われることで，指数的にプロシージャの呼び出しと逆呼び出しの回数が増えてしまい，実行時間が悪化する．

上記のメモリ使用量の効率化を行ったとしても，*rank* の可逆シミュレーション rankE をクリーンにするには，さらに，28 行目で行っている配列 p のゼロクリアをゴミスタック g を使わずに行わなくてはならない．プロシージャrankE の中でのこうした効率化は，少なくとも配列 p を再度走査することが必要になるので，clear_array の呼び出しを置き換えるだけでは効率的に実現できない．

rankE が Algol プロシージャ rankA の可逆シミュレーションになっていることは次のことから確かめられる．節点数 n の二分木 T に対応する木置換が $p[0 : n-1]$ の場合を考える．この場合，

$$[\![\texttt{rankE}]\!]^{\texttt{Janus}} : (p[0 : n-1], n) \mapsto ((n, r), g) \quad (8)$$

となり，$r = \mathit{rank}'(T)$ である．ただし，ここで常に 0 や nil をとる r と g の入力値および $p[0 : n-1]$ の各要素の出力値は省略し，元の出力値である (n, r) は組の中の組で表した．ここで，任意の n と $p[0 : n-1]$ に対して

$$\mathit{fst}([\![\texttt{rankE}]\!]^{\texttt{Janus}}(p[0 : n-1], n)) = [\![\texttt{rankA}]\!]^{\texttt{Algol}}(p[0 : n-1], n) \quad (9)$$

であるので，可逆 Janus プログラム rankE は Algol プログラム rank の可逆シミュレーションである．

5.2 埋め込みによるアンランク計算

図 4 の 42–44 行目に示されているのは，ランク r と二分木の節点数 n から木置換 $p[0 : n-1]$ を計算するプロシージャ unrankE(p,n,r,g) である．すなわち，このプロシージャはゴミスタック g を使ってランク r で節点数 n の木置換をゼロクリアされた配列 p[0:$n-1$] に格納する．同図の 11–39 行目に示されているのは，長さ n の木置換の各要素にオフセット s を足したものを部分配列 $p[a : a+n-1]$ にするプロシージャ riE である．

13 行目で呼び出されるプロシージャ clT はランク r から左部分木の大きさを計算して j に格納する．r が T のランクで呼び出された場合，r から

```
1  // 符号化されたrから|l(T)|をjに抽出
2  procedure clT(int j,int n,int r)
3  // {j=0,n=|T|,r=rank'(T)}
4    from j = 0 loop
5      r -= B[j]*B[n-j-1] // rからG_jnを減算
6      j += 1
7    until r < B[j]*B[n-j-1]
8  // {j=|l(T)|,n=|T|,r
       =rank'(l(T)) × B_|r(T)| + rank'(r(T))}
9
10 //p[a:a+n-1]にunrank(r,n)+(s,...,s)を格納
11 procedure riE(int p[],int n,int a,int s,
       int r,stack g)
12  local int j=0
13    call clT(j,n,r)        // (a) |l(T)|をjに
14
15    p[a] ^= j+s            // 根の設定
16
17    if j>1 then            // (b) l(T)の構築
18      local int r1=r/B[n-j-1] //rank'(l(T))
19        call riE(p,j,a+1,s,r1,g)
20        push(r1,g)
21      delocal int r1=0
22    else
23      if j=1 then          // |l(T)| = 1の場合
24        p[a+1] ^= s
25      fi j=1
26    fi j>1
27
28    if n-j-1 > 1 then      // (c) r(T)の構築
29      local int r2=r%B[n-j-1] //rank'(r(T))
30       call riE(p,n-j-1,a+j+1,s+j+1,r2,g)
31       push(r2,g)
32      delocal int r2=0
33    else
34      if n-j-1 = 1 then    // |r(T)| = 1の場合
35        p[a+j+1] ^= j+1+s
36      fi n-j-1 = 1
37    fi n-j-1 > 1
38    push(j,g)
39  delocal int j=0
40
41 // ランクrで節点数nの木置換を計算
42 procedure unrankE(int p[],int n,int r,
       stack g)
43   call riE(p,n,0,0,r,g)
44   push(r,g)
```

図 4　ゴミスタックを使ったランクから木置換の計算

$G_{j,|T|}$ $(j=0,1,\dots,|l(T)|-1)$ が繰り返し引かれて，j の値が $|l(T)|$ になり，r は条件 (b) もしくは条件 (c) を満たす部分木の数になる．

プロシージャ riE で使われている局所変数 j は，39 行目で解放する直前に値をゴミスタック g に移動させる必要がある．したがって，プロシージャ riE の呼び出し回数に比例した大きさのメモリがゴミスタックのために必要である．プロシージャ riE の実行後に r に格納されている値は初期のランクの値でも 0 でもない値が格納されている．この値が 44 行目においてゴミスタック g に移されることで，変数 r はクリアされる．

この埋め込みによる可逆アンランク計算をゴミスタックを使用しないクリーンなものにするには，局所的にコードを置き換えるだけでは不十分である．たとえば，unrankE の本体で push の直前で r に格納されている値は，計算の途中で出現した値であり，木置換 p の中身を詳しく見なくては可逆的に消去することができない．

前節で rankE が Algol プロシージャ rankA の可逆シミュレーションになっていることを確認したのと同じように，unrankE が Algol プロシージャ unrankA の可逆シミュレーションになっていることは次のことから確かめられる．節点数 n の二分木のランク r に対して

$$[\![\texttt{unrankE}]\!]^{\texttt{Janus}} : (n,r) \mapsto ((p[0:n-1],n),g) \quad (10)$$

となり，$T = unrank(n,r)$ に対応する木置換が $p[0:n-1]$ である．ただし，ここで常に 0 や nil をとる $p[0:n-1]$ の各要素と g の入力値および r の出力値は省略し，元の出力値は組の中の組である $(p[0:n-1],n)$ で表した．ここで，任意の n と r に対して

$$fst([\![\texttt{unrankE}]\!]^{\texttt{Janus}}(n,r)) = [\![\texttt{unrankA}]\!]^{\texttt{Algol}}(n,r) \quad (11)$$

であるので，可逆 Janus プログラム unrankE は Algol プログラム unrankA の可逆シミュレーションである．

5.3　Bennett 法による可逆シミュレーション

単射関数とその逆関数の可逆シミュレーションを組み合わせてクリーン可逆シミュレーションを実現する Bennett 法が知られている [3]．したがって，Bennett 法を用いると，単射関数 $rank$ とその逆関数 $unrank$ を実現する可逆シミュレーションである rankE と unrankE を組み合わせて，クリーン可逆シミュレーションが実現可能である．具体的には，木置換 $p[0:n-1]$ から，そのランクが r に格納され，配列 p[0:$n-1$] がゼロクリアされるという手順で構成される:

```
1 local stack g = nil
2   call rankE(p,n,r,g)
3   local int t = r
4     uncall rankE(p,n,t,g) // t,g をクリア
5   delocal int t = 0
6   local int q[n] = {0}     // 全要素を 0 に
7     call unrankE(q,n,r,g)
8     uncall copy(p,q,n)     // q をクリア
9   delocal int q[n] = {0}
10  uncall unrankE(p,n,r,g) // p,g をクリア
11 delocal stack g = nil
```

2 行目のプロシージャ`rankE` の呼び出しでゴミ情報が `g` に蓄えられつつ木置換 $p[0:n-1]$ に対応するランクが `r` に格納される．3 行目で `t` にコピーされたランクとゴミスタック `g` に蓄えられたゴミ情報は 4 行目のプロシージャ `rankE` の逆呼び出しでゼロクリアされる．7 行目のプロシージャ `unrankE` の呼び出しでゴミ情報を `g` に蓄えながら `r` に格納されたランクに対応する木置換が `q` に計算される．8 行目で配列 `p[0:`$n-1$`]` をゼロクリアされた配列 `q[0:`$n-1$`]` にコピーするプロシージャ `copy(p,q,n)` が逆呼び出しされて `q` がゼロクリアされて，10 行目のプロシージャ `unrankE` の逆呼び出しによって `p` がゼロクリアされ `g` の値が *nil* にされる．このように，局所的に使われる変数 `t`，スタック `g`, および配列 `q`, ならびに入力データである配列 `p[0:`$n-1$`]` は解放時や実行後にはすべてゼロクリアされる．

Bennett 法では，関数の可逆シミュレーションとその逆計算および逆関数の可逆シミュレーションとその逆計算がそれぞれ必要であり，出力データのコピーが必要である．また，ゴミデータや中間データを格納するメモリが必要である．これらの点で Bennett 法は時間的にも空間的にも効率が悪い．

5.4 ランク計算のクリーン可逆シミュレーション

図 3 と図 4 のプログラムは，着目する部分木の節点数が 1 のとき以外は，類似した部分がある．条件 (a), (b), (c) に対応するコード片同士で，繰り返しやプロシージャの再帰呼び出しが出現しているところが類似している．また，`rankE` で入力される木置換と出力されるランクが `unrankE` で入力されるランクと出力される木置換と一致する場合，補助プロシージャ `rE` と `riE` の呼び出し木が一致する．このように類似

$rank(T)$
$\updownarrow$ (a)
$|l(T)|, r, |T|$
$\updownarrow$ $rank'(l(T)) = r/B_{|T|-|l(T)|}$
$|l(T)|, l(T), r, |T|$
$\updownarrow$ (b)
$l(T), rank'(l(T)), rank'(r(T)), |T|$
$\updownarrow$ (c)
$l(T), l(T), r(T), |T|$
$\updownarrow$ 木置換の先頭要素が $|l(T)|$
T の木置換, $|T|$

図 5 二分木 T のランクから T の木置換の可逆的な求め方

部分を共通化することで効率的な可逆シミュレーションを得るという着想が得られる．

空木でない二分木 T の場合，(a), (b), (c) に関する計算をして図 5 のようにデータを可逆に変化させていくことができる． ただし，$r = rank'(l(T)) \times B_{|r(T)|} + rank'(r(T))$ とする．この着想を基に作成したのが図 6 にあるプログラムである．プロシージャ `unrank` は，ランクと二分木の節点数から木置換を計算する可逆シミュレーションである．この可逆シミュレーションでは最終的に出力されるゴミ情報はない．本節の以下では，T は与えられた木置換 $p[0:n-1]$ に対応する二分木とする．

実質的な計算は，ゼロクリアされた配列 `p[a:a+n-1]` にランク `r` の木置換の各要素にオフセット `s` を加算したものを格納するプロシージャ`ri(p,n,a,s,r)` によって以下のように行われる．

`n` が 0 の場合，すなわち $|T| = 0$ の場合は，空列 $p[a:a-1]$ が求めたい木置換であり，何もせず計算を終える．

`n` が 1 以上の場合，すなわち $|T| \geq 1$ の場合は，次のように再帰的に木置換を構築する．図 4 で定義されたプロシージャ `clT(j,n,r)` の呼び出しにより，符号化された `r` から $|l(T)|$ を抽出して `j` に格納し，条件 (a) を満たす二分木の数を `r` から減ずる．`r1` の領域を確保して $rank'(l(T))$ に初期化し，条件 (b) を満たす二分木の数を `r` から減じてその値を $rank'(r(T))$ にする．

```
1  // 木置換にオフセット s を加えたものを p[a:a+n-1] に
2  procedure ri(int p[],int n,int a,int s,
       int r)
3   if n>=1 then                    // |T| ≥ 1 の場合
4    local int j = 0
5      call clT(j,n,r)                      //(a)
6      local int r1 = r/B[n-j-1]            //(b)
7        r -= r1*B[n-j-1]   //{r = rank'(r(T))}
8        call ri(p,j,a+1,s,r1)
9      delocal int r1 = 0
10     call ri(p,n-j-1,a+j+1,s+j+1,r) //(c)
11     p[a] ^= j+s                   // 根を設定
12   delocal int j = p[a]-s    // j は |l(T)|
13  fi n>=1
14
15 // 節点数 n のランク r の木置換を p[0:n-1] に
16 procedure unrank(int p[],int n,int r)
17   call ri(p,n,0,0,r)
```

図 6　アンランク計算のクリーン可逆シミュレーション

このとき r の保持していた値 $rank'(r(T))$ は条件 (c) を満たす二分木の数である．プロシージャri の再帰呼び出しにより，左部分木 $l(T)$ に対応する木置換の各要素にオフセット s を加算した配列 p[a+1:a+n-j-2] を構築し，r1 をゼロクリアする．同様に，10 行目におけるプロシージャ ri の再帰呼び出しにより，右部分木 $r(T)$ に対応する木置換 $p[a+j+1:a+n-1]$ を構築し，r をクリアする．木置換の根に対応する要素 p[a] には，左部分木の節点数 $|l(T)|$ にオフセット s を加算したものが格納されている．したがって，$j = p[a] - s$ であるので，この情報を用いて j の割り当てを解放する．

プロシージャ unrank は，前節の Bennett 法の可逆シミュレーションと比べて効率が良い．なぜなら，配列 p の走査は 1 回だけであり，中間データを記憶する t や q といった局所変数やゴミデータを記憶するスタックが不要であるからである．

ランク計算は，逆変換されたプロシージャunrank の呼び出し，もしくはプロシージャunrank の逆呼び出しで実現できる．

副次的な成果であるが，unrank は文献[18] のアンランク計算をするプロシージャより使用している局所変数の数が少なく行数も少ないという点で効率的で簡潔なものになった．

5.5　提案する可逆シミュレーションの計算複雑度

本稿のアルゴリズムの基になっている Knott による非可逆なランク計算およびその逆計算[18] は，木置換の長さを n として計算時間が $O(n^2)$ であることが知られている[10]．したがって，これらの埋め込みによって作成された図 3 の rankE と図 4 の unrankE の計算時間も $O(n^2)$ である．

以下の考察から図 6 の unrank の計算時間も $O(n^2)$ である．ri の第 2 引数の値 n は構築する木置換の長さを表している．ri は長さが j と $n-j-1$ の木置換をそれぞれ 8 行目と 10 行目で構築している．また，5 行目のプロシージャclT の本体では，12 行目で解放されるときに指定される値である $|l(T)|$ の回数だけ繰り返しが行われる．ここで $0 \le |l(T)| \le n-1$ である．したがって，ri の最悪実行時間 $W(n)$ に関して

$$W(n) = \max_{0 \le j \le n-1} (W(j) + W(n-j-1) + \Theta(j)) \quad (12)$$

という漸化式が得られる．よって

$$W(n) \le \max_{0 \le j \le n-1} (W(j) + W(n-j-1)) + \Theta(n) \quad (13)$$

であるが，これは入力される配列の大きさが n のクイックソートの計算時間に関する不等式に一致する（文献[7] の 7.4.1 節）．したがって，可逆な二分木のランク計算 rank は最悪実行時間が $O(n^2)$ となる．一方，式 12 の max において常に j が最大値を取る場合，すなわちすべての節が左の子のみをもつ場合を考えると実行時間は $\Theta(n^2)$ である．したがって，unrank の最悪実行時間は $W(n) = \Theta(n^2)$ である．

クリーンな可逆プロシージャ unrank は，以下の意味で unrankA に対して最適な可逆シミュレーションである．可逆シミュレーションが，ゴミ出力が有界で，かつ任意の入力に対して元の非可逆プログラムよりも計算複雑度が同じかそれ以下で追加のメモリの大きさが入力 x の大きさ $|x|$ に対して $g(|x|)$ 以下である場合は，g によって押さえられるゴミを用いた faithful な可逆シミュレーションと呼ばれる[1]．また，g によって抑えられるゴミを用いた faithful な可逆シミュレーションよりもゴミ出力の大きさが漸近的に小さいものが存在しないとき，その可逆シミュレーションを hygienic と呼ぶ[1]．プロシージャ unrank は，ゴミ出力が無く，任意の入力に対してランク計算 unrankA と計算複雑度が同じで，追加のメモリが必

要ないので，定数関数で抑えられる hygienic な可逆シミュレーションである．

6 まとめ

Knott のランク・アンランクのプログラム [18] について一般的な Bennett 法による可逆シミュレーションよりも時間的にも空間的にも効率的なクリーン可逆シミュレーションを提案した．特に空間計算量はオーダーでも効率的になり，可逆シミュレーションは時間的にも空間的にも元の非可逆なアルゴリズムと漸近的には同じ効率になった．これは，ランク・アンランク計算をよく解析してそれらの冗長部分を融合できたことによる．

副次的な結果として元のアンランクプログラム `unrankA` よりも計算ステップ数が少ない可逆プログラムが得られたが，これはプログラムと逆プログラムから可逆プログラムを作成する際に得られる知見がより効率的なプログラム・逆プログラムの作成に役立つ可能性を示唆している．

謝辞 本研究は JSPS 科研費 25730049 の助成を受けたものである．

参 考 文 献

[1] Axelsen, H. B. and Yokoyama, T.: Programming Techniques for Reversible Comparison Sorts, *Programming Languages and Systems*, Feng, X. and Park, S.(eds.), Lecture Notes in Computer Science, Vol. 9458, Springer-Verlag, 2015, pp. 407–426.

[2] Baker, H. G.: NREVERSAL of Fortune — The Thermodynamics of Garbage Collection, *International Workshop on Memory Management. Proceedings*, Bekkers, Y. and Cohen, J.(eds.), Lecture Notes in Computer Science, Vol. 637, Springer-Verlag, 1992, pp. 507–524.

[3] Bennett, C. H.: Logical Reversibility of Computation, *IBM Journal of Research and Development*, Vol. 17, No. 6(1973), pp. 525–532.

[4] Bennett, C. H. and Landauer, R.: The Fundamental Physical Limits of Computation, *Scientific American*, Vol. 253, No. 1(1985), pp. 48–56. Bennett, C. H., and Landauer, R. (唐木幸比古訳): 計算の物理的な限界はあるか？, サイエンス, Vol. 9(1985), pp. 104–114.

[5] Bérut, A., Arakelyan, A., Petrosyan, A., Ciliberto, S., Dillenschneider, R., and Lutz, E.: Experimental verification of Landauer's principle linking information and thermodynamics, *Nature*, Vol. 483(2012), pp. 187–189.

[6] Burignat, S., Vermeirsch, K., Vos, A. D., and Thomsen, M. K.: Garbageless Reversible Implementation of Integer Linear Transformations, *Reversible Computation. Proceedings*, Glück, R. and Yokoyama, T.(eds.), Lecture Notes in Computer Science, Vol. 7581, Springer-Verlag, 2013.

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C.: *Introduction to Algorithms*, The MIT Press, 3rd edition, 2009.

[8] De Vos, A.: *Reversible Computing: Fundamentals, Quantum Computing, and Applications*, Wiley-VCH, 2010.

[9] Early, D., Gao, A., and Schellekens, M.: Frugal Encoding in Reversible $\mathcal{MOQA}$: A Case Study for Quicksort, *Reversible Computation. Proceedings*, Glück, R. and Yokoyama, T.(eds.), Lecture Notes in Computer Science, Vol. 7581, Springer-Verlag, 2013, pp. 85–96.

[10] Er, M. C.: Enumerating ordered trees lexicographically, *The Compututer Journal*, Vol. 28, No. 5(1985).

[11] Feynman, R. P.: Reversible computation and the thermodynamics of computing (chapter 5), *Feynman Lectures on Computation*, Hey, A. J. G. and Allen, R. W.(eds.), Addison-Wesley, 1996, pp. 137–184. Feynman, R. P. (Hey, A. J. G., Allen, R. W. 編, 原康夫，中山健，松田和典訳): ファインマン計算機科学, 第 5 章「可逆計算と計算の熱力学」, 岩波書店, (1999).

[12] Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A.: Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem, *ACM Transactions on Programming Languages and Systems*, Vol. 29, No. 3(2007), pp. 1–65.

[13] Frank, M. P.: *Reversibility for Efficient Computing*, PhD Thesis, Massachusetts Institute of Technology, 1999.

[14] Glück, R. and Kawabe, M.: Derivation of deterministic inverse programs based on LR parsing, *Functional and Logic Programming. Proceedings*, Kameyama, Y. and Stuckey, P. J.(eds.), Lecture Notes in Computer Science, Vol. 2998, Springer-Verlag, 2004, pp. 291–306.

[15] Gries, D.: *The Science of Programming*, Texts and Monographs in Computer Science, Springer-Verlag, 1981, chapter 21 Inverting Programs, pp. 265–274.

[16] Gruska, J.: *Quantum Computing*, McGraw-Hill, 1999.

[17] James, R. P. and Sabry, A.: Information effects, *Principles of Programming Languages. Proceedings*, ACM Press, 2012, pp. 73–84.

[18] Knott, G. D.: A numbering system for binary trees, *Communications of the ACM*, Vol. 20, No. 2(1977), pp. 113–115.

[19] Knuth, D. E.: *The Art of Computer Program-*

ming, Volume 1: Fundamental Algorithms, Vol. 1, Addison-Wesley Professional, 3rd edition, 1997.

[20] Landauer, R.: Irreversibility and Heat Generation in the Computing Process, *IBM Journal of Research and Development*, Vol. 5, No. 3(1961), pp. 183–191.

[21] Landauer, R.: Information is Physical, *Physics Today*, Vol. 44, No. 5(1991), pp. 23–29. Landauer, R. (小柳義夫訳): 情報は物理過程だ, パリティ, Vol. 6, No. 11(1991), pp. 30–39.

[22] Lutz, C.: Janus: A time-reversible language, 1986. Letter to R. Landauer.

[23] Mogensen, T. Æ.: Garbage-Free Reversible Multipliers for Arbitrary Constants, *ACM Journal on Emerging Technologies in Computing Systems*, Vol. 11, No. 2(2014), pp. 1–18.

[24] Nishida, N., Sakai, M., and Sakabe, T.: Partial Inversion of Constructor Term Rewriting Systems, *Term Rewriting and Applications. Proceedings*, Giesl, J.(ed.), Lecture Notes in Computer Science, Vol. 3467, Springer-Verlag, 2005, pp. 264–278.

[25] Perumalla, K. S.: *Introduction to Reversible Computing*, CRC Press, 2013.

[26] Pesu, T. and Phillips, I.: *Real-Time Methods in Reversible Computation*, Springer International Publishing, 2015, pp. 45–59.

[27] Saeedi, M. and Markov, I. L.: Synthesis and Optimization of Reversible Circuits — A Survey, *ACM Computing Survey*, Vol. 45, No. 2(2013), pp. 21:1–21:34.

[28] Sprugnoli, R.: The Generation of Binary Trees as a Numerical Problem, *Journal of the ACM*, Vol. 39, No. 2(1992), pp. 317–327.

[29] Thomsen, M. K., Axelsen, H. B., and Glück, R.: A Reversible Processor Architecture and Its Reversible Logic Design, *Reversible Computation. Proceedings*, De Vos, A. and Wille, R.(eds.), Lecture Notes in Computer Science, Vol. 7165, Springer-Verlag, 2012, pp. 30–42.

[30] Wille, R. and Drechsler, R.: *Towards a Design Flow for Reversible Logic*, Springer-Verlag, 2010.

[31] Yokoyama, T., Axelsen, H. B., and Glück, R.: Principles of a Reversible Programming Language, *Computing Frontiers. Proceedings*, ACM Press, 2008, pp. 43–54.

[32] Yokoyama, T., Axelsen, H. B., and Glück, R.: Optimizing Clean Reversible Simulation of Injective Functions, *Journal of Multiple-Valued Logic and Soft Computing*, Vol. 18, No. 1(2012), pp. 5–24.

[33] Yokoyama, T., Axelsen, H. B., and Glück, R.: Towards a Reversible Functional Language, *Reversible Computation. Proceedings*, De Vos, A. and Wille, R.(eds.), Lecture Notes in Computer Science, Vol. 7165, Springer-Verlag, 2012, pp. 14–29.

[34] Yokoyama, T. and Glück, R.: A Reversible Programming Language and Its Invertible Self-Interpreter, *Partial Evaluation and Semantics-Based Program Manipulation. Proceedings*, ACM Press, 2007, pp. 144–153.

[35] 森田憲一: 可逆計算, 近代科学社, 2012.

A 補助プロシージャ

本章では，本稿中に用いた補助プロシージャを示す.

可逆プロシージャ`clear_array`は，配列`p[0:`$n-1$`]`の値をゴミスタック`g`に移す.

```
1 procedure clear_array(int p[],int n,
     stack g)
2   local int i = 0
3     from i = 0 do
4       local int t = p[i]
5         p[i] ^= t // クリア
6         push(t,g)
7       delocal int t = 0
8       i += 1
9     until i = n
10  delocal int i = n
```

可逆プロシージャ`copy`は，`x[0:`$n-1$`]`を`y[0:`$n-1$`]`に排他的論理和の複合代入演算子を用いて可逆的にコピーする．したがって，`x[0:`$n-1$`]`と`y[0:`$n-1$`]`の全要素が同一の値を格納している場合は`y[0:`$n-1$`]`をゼロクリアする．`copy`は呼び出しと逆呼び出しとで振る舞いは変わらない.

```
1 procedure copy(int x[],int y[],int n)
2   local int i = 0
3     from i = 0 do
4       y[i] ^= x[i]
5       i += 1
6     until i = n
7   delocal int i = n
```