2 次元可逆分割セルオートマトンの可逆プログラミング言語上でのクリーン可逆シミュレーション

M2018SE013 矢澤拓海
指導教員：横山哲郎

1 はじめに

セルオートマトン（以下，CA）は結晶の成長などの化学システムや交通モデルなどの社会システムといった様々な研究分野においてシミュレーションに用いられている計算モデルである．CA の状相（セル空間全体の状態）の遷移を行う大域関数に単射性の制約を与え，前の時間の状相が一意に定まるものが可逆 CA（以下，RCA）である．可逆計算は計算に関わる消費エネルギーに深い関係があり，可逆 Turing 機械（以下，RTM）などの計算モデルや観測以外の演算が可逆性制約を満たす必要がある量子計算など様々な応用がある．非可逆の場合と同様に，可逆な計算モデルの表現能力の解析のために様々な可逆シミュレーションが実現されている．例えば，RTM の可逆プログラミング言語による可逆シミュレーションが知られている．

本研究では，RCA を可逆プログラミング言語 Janus[1] で可逆シミュレーションする．これまで周期的境界条件をもつ 1 次元 RCA（以下，1D-RCA）[2] や 1 次元可逆分割 CA（以下，1D-RPCA）[3] がスタック付き Janus[4]（以下，単に Janus）で実現されている．さらに，1D-RPCA は状相のうち非静止状態がある範囲のみのセルの情報を直接的にスタックにもつだけで Janus で可逆シミュレーションできる [5].

本研究では，2D-RPCA を状相のうち非静止状態がある矩形範囲のみのセルの情報を直接的に 2 次元配列でもつことで可逆シミュレーションする．この実現には 2 次元配列を実行時に可逆的に割付け/解放できる Janus を用いる．2D-RPCA の大域関数は単射であるにもかかわらず，状相の表現の効率化を行う場合，上記のように状相の情報をもつ 2 次元配列の更新は必ずしも単射にはならない問題がある．我々は状相をメモリ使用量を最適にした可逆動的表で保持して Bennett の入力消去可逆シミュレーション [6, 7] を応用することでこの問題を解決する．我々が知る限りでは 2D-RPCA の可逆シミュレーションは可逆プログラミング言語で効率的に実現されていない．2D-RPCA の可逆シミュレーションの Janus での実現は，Janus の可逆 Turing 完全性の別証明である．

2 関連研究

可逆プログラミング言語 Janus[1] は【実行時に割り付けた】有限のメモリしか扱うことができず r-Turing 完全ではなかった．文献 [4] において，Janus 上で可逆スタックを用いることにより，RTM のクリーン可逆シミュレーションを行っていた．これにより，可逆スタックのある Janus が r-Turing 完全であることの証明がされている．Janus 上での 1D-RPCA のクリーン可逆シミュレーションの実現については，複数の方法が提案されている．文献 [2] において，周期的境界条件を持つ 1D-PRCA が実現されている．また，文献 [3] は，非周期境界条件をもつ 1 次元 RCA のクリーン可逆シミュレーションをスタック付きの Janus で実現した．この実現方法では状相の遷移の度にスタックが大きくなり，メモリ使用量がだんだん大きくなる問題があった．文献 [5] の 1D-RPCA では，状相の表現の効率化により，無限に広がる状相を扱える 1D-RPCA の実現がされている．

3 準備

可逆計算とは，全ての時間の状態から直前の状態がたかだか一つに特定できるような計算のことをいう．たとえばある計算を単射の写像として表せるならばその計算は可逆である．本章では以降の章で用いる可逆性をもつ CA とプログラミング言語を定める．

以降では，$\bar{t}$ が文脈によって組 $(t_1, \ldots, t_m)$ 又は積 $t_1 \times \cdots \times t_m$ を表すこととする．

3.1 CA と RCA

CA は有限オートマトンであるセルを規則的に配置・接続した離散計算モデルである．各セルは近傍の現在の状態によってのみ次の状態が決まる．特に各セルが複数部分からなる k 次元分割 CA（以下，kD-PCA）が知られている．

kD-PCA は $(\mathbb{Z}^k, \overline{Q}, \overline{n}, f, \overline{\#})$ と定められる．ここで，$\mathbb{Z}^k$ はセルが配置される k 次元ユークリッド空間の整数座標をもつ点の集合である．各セルは m 個の部分からなる．$Q_i (i = 1, \ldots, m)$ は，各セルの第 i 部分のとり得る内部状態の非空有限集合である．m 個組 $\overline{n}$ をセルの近傍とよぶ．関数 $f : \overline{Q} \to \overline{Q}$ は各セルの状態遷移を表す局所関数である．$\overline{\#} \in \overline{Q}$ は，$f(\overline{\#}) = \overline{\#}$ を満たす静止状態である．

単射大域関数をもつ CA を可逆とよぶ．可逆 PCA では局所関数と全域関数の全単射性は等価である．

3.2 可逆プログラミング言語 Janus

本節では，可逆プログラミング言語 Janus について記述する．Janus は C に似た構文をもつ手続き型の可逆プログラミング言語であり，単射な意味をもつ文のみ記述できるという意味で可逆である．Janus では単射でない単純な代入はできず，代わりに加算，減算，排他的論理和の 3 種類の複合代入演算子で代入を行う．局所変数と 2 次元配列の割付けと値の初期化は `local` で行う．`local` で割付けを

した局所変数と 2 次元配列は delocal で保持する値を指定して解放を行う必要がある．非可逆言語の if 文では，その実行後に then 節と else 節のどちらに分岐していたか判別できない．この制御の更新は単射ではなく可逆性が失われる．Janus では if 文の末尾にアサーションをもち，どちらの節に分岐が行われたかが if 文の実行直後に一意に定まるようにする．このアサーションは，then 節に分岐した場合は真を表す値を，else 節に分岐した場合は偽を表す値をもたねばならない．非可逆言語の繰返しに入る制御の合流地点においても，繰返しが開始されたのか，1 回以上繰返しが行われているのかが特定できないので可逆性が失われる．Janus では繰返し文の先頭にアサーションをもち，繰返しの内部と外部のどちらから制御がきたかを一意に定まるようにする．このアサーションは，繰返しの開始時に真を表す値を，1 回以上繰返した後である場合は偽を表す値をもたねばならない．Janus では call を行うことにより，プロシージャを呼び出すことができる．uncall はプロシージャを逆方向に実行する逆呼出しをする．スタックへのプッシュは，push(x,g) を用いて行う．スタック g に変数 x の値を格納し，その後，x の値を 0 クリアする．pop は pop(x,g) を用いて行い，スタック g の先頭の値を取り出し x に格納する．このとき，変数 x の値は 0 でなければならない．push と pop は互いに逆プロシージャである．

本研究では Janus に 2 次元配列と配列の割付け/解放が独立して行える拡張を行ったものを用いる．構文領域を次のように拡張する：

$$
\begin{array}{lll}
d ::= \cdots \mid x\texttt{[}c\texttt{]}\texttt{[}c\texttt{]} & & \text{スカラと配列} \\
e ::= \cdots \mid x\texttt{[}e\texttt{]}\texttt{[}e\texttt{]} & & \text{式} \\
s ::= \cdots \mid x\texttt{[}e\texttt{]}\texttt{[}e\texttt{]} \oplus\texttt{=}\ e \mid & & \text{文} \\
\qquad \texttt{local}\ t\ x\ \texttt{=}\ e \mid \texttt{delocal}\ t\ x\ \texttt{=}\ e & &
\end{array}
$$

状態を σ，値を v と書くことにする．式と文の判定をそれぞれ $\sigma \vdash_{expr} e \Rightarrow v$ と $\sigma \vdash_{stmt} s \Rightarrow \sigma$ とする．割付けと解放の自然意味論を定める：

$$
\text{Local}\ \frac{x \notin \mathrm{dom}(\sigma) \quad \sigma \vdash_{expr} e_1 \Rightarrow v_1 \quad \sigma \vdash_{expr} e_2 \Rightarrow v_2}{\sigma \vdash_{stmt} \texttt{local}\ t\ x\texttt{[}e_1\texttt{]}\texttt{[}e_2\texttt{]}\ \texttt{=}\ \texttt{\{\{0\}\}} \Rightarrow \sigma[x\texttt{[0][0]} \mapsto 0] \cdots [x\texttt{[}v_1\texttt{]}\texttt{[}v_2\texttt{]} \mapsto 0]}
$$

$$
\text{Delocal}\ \frac{x \notin \mathrm{dom}(\sigma) \quad \sigma \vdash_{expr} e_1 \Rightarrow v_1 \quad \sigma \vdash_{expr} e_2 \Rightarrow v_2}{\sigma[x\texttt{[0][0]} \mapsto 0] \cdots [x\texttt{[}v_1\texttt{]}\texttt{[}v_2\texttt{]} \mapsto 0] \vdash_{stmt} \texttt{delocal}\ t\ x\texttt{[}e_1\texttt{]}\texttt{[}e_2\texttt{]}\ \texttt{=}\ \texttt{\{\{0\}\}} \Rightarrow \sigma}
$$

delocal 文における 2 次元配列の解放は，正しい配列サイズが指定され，全要素が 0 をもつ場合にのみ成功する．本研究で用いる Janus においては local と delocal は入れ子の形で用いる必要は無い．このような拡張を行った Janus も可逆である．

プリミティブ関数 size(x) は配列 x の大きさを返す．

```
procedure CA(int t_end, stack conf, int rule[][])
  iterate int t = 0 to t_end
      call global_map(conf, rule)
  end
```

図 1 1D-RPCA のクリーン可逆シミュレーション

```
procedure global_map(stack sr, int rule[][])
 local stack sl=nil,int l=0,int c=0,int r=0,int next_r=0
 call mypop(l, sr)
 from empty(sl) && c=0 && r=0 && next_r=0 loop
  call local_map(l, c, r, rule)
  call mypush(l,sl) call mypush(c,sl) call mypush(r,sl)
  call mypop(c,sr)  call mypop(r,sr)  call mypop(l,sr)
  r <=> next_r
 until empty(sr) && l=0 && c=0 && r=0 && next_r=0
 call rewind(sl, sr)
 delocal stack sl=nil,int l=0,int c=0,int r=0,int
      next_r=0
```

図 2 大域関数

4 1D-RPCA の可逆シミュレーション

本章では，文献 [5] で実現された 1D-RPCA[9] の可逆プログラミング言語 Janus 上での実現方法を示す．文献 [5] で実現された 1D-PRCA は，状相の表現を，有限個の静止状態でない状態が含まれる範囲のみにすることで，大域関数の単射性を保ちつつ，効率的な表現にしている．1D-PRCA は RTM をシミュレートできるので，このプログラムは Janus の r-Turing 完全性の証明になっている．

4.1 1 次元 3 近傍分割セルオートマトン

本節では，今回実装する 1D-RPCA について記述する．1D-RPCA $P = (\mathbb{Z}, (L, C, R), f, (\#_1, \#_2, \#_3))$ を定める．ここで，$\#_1 = \#_2 = \#_3 = 0$, $L = C = R = \{0, 1\}$ であり，f は表 1 で定まる $f(l, c, r) = (l', c', r')$ の場合で，P は有限状相のみをもつものとする．

4.2 可逆プログラム

表 1 1D-RPCA の局所関数 $f(l, c, r) = (l', c', r')$

lcr	$l'c'r'$	lcr	$l'c'r'$
000	000	100	100
001	001	101	101
010	010	110	110
011	011	111	111

図 1 に時間発展に伴う P の状相の変化を計算する CA(t_end,conf,rule) を示す．ただし，t_end は終了ステップ数を表し，スタック conf は状相を表すものであり，スタックの底には必ず非静止状態が格納され，かつセルの分割位置が変わらないよう静止状態の個数が最適化されている．【な最小数の静止状態が用いられている．】2 次元配列 rule[no][j] は表 1 の左上から右下の no 番目の l' (j = 0), c' (j = 1), 及び r' (j = 2) の値をそれぞれ表す．【j = 1, 2, 3 のときの rule[no][j] は表 1 の左上から右下の no 番目の l', c', 及び r' の値をそれぞれ表す．】

図 2 に大域関数を計算する可逆プロシージャ global_map を示す．このプロシージャは，局所関数 local_map を用いて，ループするごとにスタックで表現されたセルの回数ぶん繰り返し計算する．各ループで

```
procedure local_map(int l, int c, int r, int rule[][])
  local int no = (l<<2) + (c<<1) + (r<<0)
    l ^= ((no&0x4)>>2) ^ rule[no][0]
    c ^= ((no&0x2)>>1) ^ rule[no][1]
    r ^= ((no&0x1)>>0) ^ rule[no][2]
    from l = rule[no][0] &&
         c = rule[no][1] &&
         r = rule[no][2] loop
      no -= 1 // zero clear no
    until no = 0
  delocal int no = 0
```

図 3 局所関数

```
procedure rewind(stack sl, stack sr)
  call mypush(0, sr) // the last l
  from empty(sr) do
   local int t = 0
    call mypop(t,sl) call mypush(t,sr) //move R cell
    call mypop(t,sl) call mypush(t,sr) //move C cell
    call mypop(t,sl) call mypush(t,sr) //move L cell
   delocal int t = 0
  until empty(sl)
```

図 4 スタックの移し替え

は，l, c, r の値を用いて 局所関数を実行し，後にスタック sl にセルの 3 要素を順に格納する．次に近傍に相当する位置の，着目しているセルの C の要素，直後のセルの R の要素，及び手前のセルの L の要素をスタック sr から取り出してそれぞれ変数 c, r, 及び l にこの順に格納する．r は 1 回遅らせて利用するため，next_r と値をスワップする．

ループの前では静止状態ではない最左のセルの直ぐ左のセルに着目する．すなわち，sr から取り出した要素を l に格納して，c = r = 0 という条件を満たしたままループに入る．ループ終了後は，可逆プロシージャ rewind によって sl に格納されたセルを 3 つずつ逆順に sr に移す．要素を 3 つずつ格納することにより，分割位置を保つ．

図 3 に局所関数 f を計算する local_map を示す．まず l, c, r の値を利用してその状態の際に適用するのルールの番号を計算して no に格納する．3 行目から 5 行目では，それぞれの変数を no を用いてゼロクリアしてルールで変更された状態に更新する．ループでは，書き換え後の状態の情報を用いて no をゼロクリアする．

図 4 にスタック sl の要素を逆順にスタック sr に移す rewind(sl,sr) を示す．4 行目では，注目している直後のセルにおける L に対応する要素をプッシュする．その後，ループを繰り返すたび，3 つずつ要素を移していく．これにより，セルの分割位置が変化することがないように移すことができる．

図 5 にこれまで使ってきたプッシュとポップの定義を示す．無限に広がる 1 次元の状相を表現する際，文献 [4] で行われた RTM の無限長のテープを有限で表す方法を応用することができる．我々は文献 [4] と同様に図 5 のような push と pop の拡張を行った．空のスタックに静止状態を push する際には何も入れず，pop は push の逆呼出しを用いて定義する．

```
procedure mypush(int x, stack s)
  if !(empty(s) && x=0) then
    push(x, s)
  fi !empty(s)

procedure mypop(int x, stack s)
  uncall mypush(x, s)
```

図 5 拡張したプッシュとポップ

	u	0		1	
l	r\d	0	1	0	1
0	0	0000	0100	0001	1010
	1	0010	1001	1100	1101
1	0	1000	0011	1001	0111
	1	0101	1011	1110	1111

【森田先生とは違う表現でかつ少ないスペースで表したい...】

$urdl$	$u'r'd'l'$	$urdl$	$u'r'd'l'$	$urdl$	$u'r'd'l'$	$urdl$	$u'r'd'l'$
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000

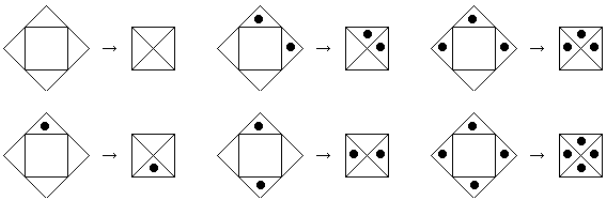

図 6 2D-RPCA P の遷移規則

可逆プロシージャ CA(t,conf,rule) の空間計算量を考える．conf が保持するのは，非静止状態のセルを両端とする必ずしも静止状態ではないセルが存在する範囲である．これにより，無限個の静止状態のセルを有限メモリで表現できるので，提案プロシージャのメモリ使用量は小さい．

5 2D-RPCA の可逆シミュレーション

本章では，2D-RPCA とその可逆シミュレーションの方法を述べる．

5.1 2 次元 4 近傍分割セルオートマトン

2D-PCA $P = (\mathbb{Z}^2, \overline{Q}, \overline{N}, f, \overline{\#})$ を定める．ここで，近傍は $\overline{N} = ((0,-1),(-1,0),(0,1),(1,0))$，$Q_i = \{0,1\}$，$\#_i = 0$ $(i = 1,2,3,4)$，f は図 6 に示す遷移規則とそれらをそれぞれ 90°, 180°, 270° 回転させたものの集合である．遷移規則の右辺が全て異なるので，P は可逆である．

5.2 2 次元配列による有限状相の表現

図 7 分割された各セルの係数

P の状相を 2 次元可逆動的配列（以下単に 2 次元配列）c で表す．分割されたセルの状態 Q_i を 0/1 で表し，それらと図 7 の対応する数との積の和で効率的に状態を表す．2 次元配列の端以外にあるセル c[x][y] の近傍は c[$x \pm 1$][$y \pm 1$] というように直接的に表せる．無限のセ

```
procedure CA(int t_end, int conf[][], int rule[][])
    iterate int t = 0 to t_end
        call global_map(conf, rule)
    end
```

図 8 2D-RPCA P の t_end ステップの時間発展

ルをもつ状相を c で直接的に表すことはできない．我々は 1 次元 CA の有限状相をスタックで表現した方法 [5] を応用し，2 次元 CA の有限状相のセルが存在しうる矩形領域のみを c で直接的にもつ．

5.3 大域関数の実現

状相から状相への P の大域関数 F は定義より単射である．しかし，c を更新する計算は単射では無い．なぜなら，上記の矩形領域の取り方により境界に静止状態をもつ無数の 2 次元配列は同一の状相を表し，c を更新する計算は多対多の関係を表すからである．

単純な解決法として c を更新するときに必ず周囲に 1 要素ずつ静止状態を追加することで計算を単射にすることが考えられる．この方法だと実装が比較的簡単になる．

別の解決法も考えられる．境界に非静止状態をもつ 2 次元配列を整形とよぶ．F を表す計算を 2 次元配列の整形性を保つように入力消去可逆計算 [6, 7] で実現できる．入力消去可逆計算の実現には非可逆な計算とその逆計算の可逆的な実現が必要である．その両者を整形の c からその矩形領域の周囲にたかだか 1 要素ずつ追加した非整形の c' へそれぞれ F と F' を表す計算で実現できる．しかし，この方法では実装が複雑になるという欠点がある．

5.4 可逆プログラミング言語による実現

2D-RPCA P の可逆シミュレーションは図 8 のプロシージャ CA により行う．CA は初期状相 conf に大域関数 global_map を t_end 回適用して更新する．

図 9 に大域関数を計算する global_map を示す．まず遷移後の状相の表現を一時保存するための 2 次元配列 sl を割り付ける．次に遷移前の状相の表現をもつ sr の上下左右の境界をたかだか 1 セルずつ拡張する．続く繰返しでは各セルに対してその近傍をプロシージャ get_ulrd で一時変数 u, l, r, d に移し，それらを引数としたプロシージャ local_map で局所関数をクリーン可逆シミュレーションした結果をそれぞれ sl に格納する．プロシージャ move2 は sl と全要素が 0 クリアされた sr の要素を入れ替える．

図 10 に局所関数を計算するプロシージャ local_map を示す．近傍の値から適用する遷移規則の番号 no が一意に定まる．遷移前の近傍を遷移後の近傍に書き換え，繰返しを利用することにより使用した遷移規則の番号 no を逆算し，ゼロクリアする．遷移後のセルを t に格納し，t を用いて遷移後の近傍もゼロクリアする．

```
procedure global_map(int sr[][], int rule[][])
    local int sl[size(sr) + 2][size(sr) + 2] = {{0}},
        int u = 0, int l = 0, int r = 0, int d = 0
    call table_expand(sr)
    iterate int y = 0 to size(sl) - 1
        iterate int x = 0 to size(sl) - 1
            call get_ulrd(u,l,r,d,sr,x,y)
            local int t = 0
                call local_map(t, u, l, r, d, rule)
                t <=> sl[x][y]
            delocal int t = 0
        end
    end
    call move2(sl, sr)
    delocal int sl[size(sr)][size(sr)] = {{0}}, int u
        = 0, int l = 0, int r = 0, int d = 0
```

図 9 大域関数

```
procedure local_map(int t, int u, int l, int r, int d,
    int rule[][])
    local int no = (l<<3) + (d<<2) + (r<<1) + (u<<0)
    l ^= ((no & 0x8) >> 3) ^ rule[no][0]
    d ^= ((no & 0x4) >> 2) ^ rule[no][1]
    r ^= ((no & 0x2) >> 1) ^ rule[no][2]
    u ^= ((no & 0x1) >> 0) ^ rule[no][3]
    from l = rule[no][0] &&    // zero clear no
         d = rule[no][1] &&
         r = rule[no][2] &&
         u = rule[no][3] loop
        no -= 1
    until no = 0
    t ^= (l<<3) + (d<<2) + (r<<1) + (u<<0)
    l ^= (t & 0x8) >> 3
    d ^= (t & 0x4) >> 2
    r ^= (t & 0x2) >> 1
    u ^= (t & 0x1) >> 0
    delocal int no = 0
```

図 10 局所関数

```
{{0, 12},   {{0, 0, 0, 0},  {{0, 0, 0, 0, 0, 0},
  3, 0}}     {0, 0, 12, 0},  {0, 0, 0, 0, 0, 0},
             {0, 3, 0, 0},   {0, 0, 6, 0, 0, 0},
             {0, 0, 0, 0}}   {0, 0, 0, 9, 0, 0},
                             {0, 0, 0, 0, 0, 0},
                             {0, 0, 0, 0, 0, 0}}
```

図 11 左から初期状態，遷移前，遷移後の 2 次元配列

5.5 サンプル入出力

図 11 に状相を表す 2 次元配列の遷移を示す．遷移の際に 2 次元配列の大きさが縦横 2 ずつ増大する．要素数 n^2 の 2 次元配列からの遷移において，メモリ使用量は $4n+4$ 増え，計算ステップ数は $O(n^2)$ である．

5.6 状相の表現の効率化

5.4 節で提案した 2D-PRCA の実装方法では，5.5 節で示した通りメモリ使用量が遷移ごとに必ず増大する欠点があった．本節では，入力消去可逆計算を用いて状相を表現する配列の大きさの効率化を行う方法を提案する．

5.6.1 効率的なセル空間の表現方法

本節では，状相の効率的な表現方法を示す．状相の効率的な表現とは，ある有限状相 α を行と列の大きさが同じ 2 次元配列で表現した際に，配列の先頭の行と列，末尾の行と列に静止状態でない状態が入っている配列の表現のこと

を言う．セル空間を例として，図 11 の初期状態のような配列と，遷移前の配列は同じ状相を表している．これらの異なった表現をされた状相は同じ状相であり，大域関数を適用すると遷移後の状相が全て同じになってしまい，単射性が失われてしまう．5.4 節ではこれらの状相の表現を別のものとして扱うことで，遷移後の状相もそれぞれ別の表現になるようにし，単射性を保っていた．本節では図 11 の初期状態の配列のように，先頭の行と列，末尾の行と列に必ず静止状態以外の状態が格納されるように統一することで単射性を保つ．このようにして，状相の効率的な表現を行う．

5.6.2 効率的な表現への変換

本節では，5.4 節で示したプログラムの問題点である，状相の遷移の度にセル空間の表現が必ず拡大していくという問題に対して，解決案を提示する．効率的な遷移前後の状相をそれぞれ α_1, β_1, 非効率な遷移前後の状相をそれぞれ α_2, β_2 として，これらの状相の変換が可逆的に行うことができるかについて考察する．今回実装したいプログラムは α_1 から β_1 への遷移であり，これは，2D-RPCA の定義から単射であり，可逆である．よって，入力消去可逆計算 [6, 7] を応用し，ゴミ出力を消去することが可能である．なお，ここでは状相の表現に必要な出力以外の出力をゴミ出力とみなす．5.4 節で示したプログラムは，入力を α_1 に限定することで，α_1 から β_2 への遷移を行うゴミ出力を伴わないプログラムとみなすことができる．このように遷移された β_2 からゴミ出力を伴いながら β_1 に変換を行っても，入力消去可逆計算を利用してゴミ出力が消去可能，つまり，α_1 から β_1 へのゴミ出力が伴わない遷移を行うプログラムが実現可能である．

6 むすび

本研究では，可逆動的表を用いた 2D-RPCA の Janus 上での可逆シミュレーションと，状相の表現の効率化の手法について検討を行った．我々は，非静止状態のセルが存在する矩形領域を直接的に表すメモリのみで 2D-RPCA の可逆シミュレーションを実現する方法を示した．これは Janus の r-Turing 完全性を証明したことになる．しかし，実装上の問題が解決できず，2D-RPCA の可逆シミュレーションは遷移の度にメモリ使用量が増えるものとなった．2D-RPCA の Janus 上でのより効率的な可逆シミュレーションと他の実現方法の模索は今後の課題である．

参考文献

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

[2] Moriyama, K.: Reversible Cellular Automata from a Programming Language Perspective, Master's thesis, DIKU, University of Copenhagen (2010).

[3] 渡邉恭平：可逆スタックを用いた可逆セル・オートマトンのクリーン可逆シミュレーション，南山大学 2013 年度卒業論文 (2014).

[4] Yokoyama, T., Axelsen, H.B. and Glück, R.: Principles of a Reversible Programming Language, *Proc. CF*, ACM Press, pp.43–54 (2008).

[5] 木村孝大，矢澤拓海：1 次元可逆セル・オートマトンのクリーン可逆シミュレーションの実現，南山大学 2017 年度卒業論文 (2018).

[6] Bennett, C.H.: Logical Reversibility of Computation, *IBM J. Res. Dev.*, Vol.17, No.6, pp.525–532 (1973).

[7] Bennett, C.H.: Time/space trade-offs for reversible computation, Vol.18, No.4, pp.766–776 (1989).

[8] Morita, K.: Computation-universality of one-dimensional one-way reversible cellular automata, *IPL*, Vol.42, No.6, pp.325–329 (1992).

[9] Morita, K. and Harao, M.: Computation Universality of One-Dimensional Reversible (Injective) Cellular Automata, *Trans. IEICE Japan*, Vol.E72, No.6, pp.758–762 (1989).