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，直前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．たとえば，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，2 次元可逆分割セルオートマトン（以下，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，状相の整形な表現のみを用いた場合，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．我々は論文 [5] を応用し，1 次元での無限長の状相を有限なメモリで扱う手法を 2 次元に拡張する．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無限長の状相を有限なメモリで表現する．また，入力消去可逆シミュレーション [6, 7] を応用することで効率的なメモリ使用量での実現を図る．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 r-Turing 完全性の別証明である．また，可逆性は物理制約に深い関わりがあり，可逆シミュレーションを実現することは，様々な物理学のシミュレーションを実現する際の基礎となる．

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．可逆シミュレーションにおいてクリーンであるとは，入出力以外のゴミ情報が出力されないことである．文献 [2] において，周期的境界条件をもつ 1D-RPCA が実現されている．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ごとにメモリ使用量が増加する問題があった．文献 [5] の 1D-RPCA では，状相の整形な表現のみを用いることによりメモリ使用量を最適化した．

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，$\overline{t}$ 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．各セルは近傍の状態によってのみ直後の状態が決まる．特に各セルが複数部分からなる k 次元可逆分割 CA（以下，kD-RPCA）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．$\mathbb{Z}^k$ 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．写像 $\alpha : \mathbb{Z}^k \to \overline{Q}$ は状相である．状相全ての集合を $Conf(\overline{Q})$ で表す．集合 $\{x \mid x \in \mathbb{Z}^k \wedge \alpha(x) \neq \overline{\#}\}$ が有限であるような状相 α を有限状相という．関数 pr_i $(i = 1, \ldots, m)$ を，任意の $\overline{q} \in \overline{Q}$ に対して，$pr_i(\overline{q}) = q_i$ となるような射影関数とする．局所関数 f から誘導される大域関数 $F : Conf(\overline{Q}) \to Conf(\overline{Q})$ を $F(\alpha)(x) = f(pr_1(\alpha(x + n_1)), \ldots, pr_m(\alpha(x + n_m)))$ とする．

単射大域関数をもつ CA を可逆とよぶ．RPCA 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

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できる．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 文により，プロシージャを呼び出すことができる．uncall 文はプロシージャの本体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．ポップは `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：

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

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：

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

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

```
procedure CA(int t_end, stack conf, int rule[][])
  iterate int t = 1 to t_end
    call gmap(conf, rule)
  end
```

図 1 1D-RPCA P_1 のクリーン可逆シミュレーション

`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．Janus は単射な文のみを記述することにより可逆性を保っている．独立した `local` と `delocal` はそれぞれ単射であるため，可逆性は保たれる．`local` と `delocal` を正しく用いないことによるプログラムの異常終了が起こる可能性が考えられるが，異常終了と可逆性に直接的な関係はないため，可逆性は損なわれず，このような拡張を行った Janus も可逆である．以降では拡張された Janus を単に Janus と呼ぶ．

プリミティブ関数 `size(`x`)` は配列 x の大きさを返す．

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 Janus 上での実現方法を示す．文献 [5] で実現された 1D-RPCA は，有限個の非静止状態が含まれる状相の範囲のみを直接的にもつ整形スタックを用いることで大域関数の単射性を保ちつつメモリ使用量を最適化した．1D-RPCA は RTM を可逆的に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 次元分割セルオートマトン

各セルを 3 分割した 1D-RPCA $P_1 = (\mathbb{Z}, (L, C, R), f, (\#_1, \#_2, \#_3))$ を定める．ここで，$\#_1 = \#_2 = \#_3 = 0$, $L = C = R = \{0, 1\}$ であり，f は表 1 で定まる局所関数であり，P_1 は有限状相のみをもつものとする．

表 1 1D-RPCA P_1 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

4.2 可逆プログラム

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な最小数の静止状態が用いられている．`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', 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 `gmap` を示す．`gmap` では，繰返しごとにスタックで表現されたセル数回，局所関数 f を計算する `lmap` が呼び出される．各繰返しでは，スタックから取り出した `l`, `c`, `r` の値を用いて局所関数を実行し，`l`, `c`, `r` の値を更新する．後にスタック `sl` にセルの 3 つの部分を順に格納する．

```
procedure gmap(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 lmap(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　1D-RPCA P_1 の大域関数

```
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 3　拡張したプッシュとポップ

表 2　2D-RPCA P_2 の局所関数 $f(u,r,d,l)=(u',r',d',l')$

$urdl$	$u'r'd'l'$	$urdl$	$u'r'd'l'$	$urdl$	$u'r'd'l'$	$urdl$	$u'r'd'l'$
0000	0000	0100	0100	1000	1000	1100	0011
0001	0001	0101	1010	1001	0110	1101	0111
0010	0010	0110	1001	1010	0101	1110	1011
0011	1100	0111	1101	1011	1110	1111	1111

繰返し終了後は，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．

図 3 に `mypush` と `mypop` の定義を示す．無限に広がる 1 次元の状相を表現する際，RTM の無限長のテープを有限メモリで表す方法 [4] を応用することができる．我々は文献 [4] と同様に図 3 のようなプッシュとポップの拡張を行った．空のスタックに静止状態をプッシュする際には何も入れず，`mypop` は `mypush` の逆呼出しを用いて定義する．

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 次元分割セルオートマトン

2D-RPCA $P_2=(\mathbb{Z}^2,\overline{Q},\overline{N},f,\overline{\#})$ を定める．ここで，近傍は $\overline{N}=((0,-1),(-1,0),(0,1),(1,0))$，$Q_i=\{0,1\}$，$\#_i=0$ $(i=1,2,3,4)$，$f\,(u,r,d,l)=(u',r',d',l')$ は表 2 に示す遷移規則の集合である．f の右辺が全て異なるので単射であり，P_2 は可逆である．

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

図 4　2D-RPCA P_2 のクリーン可逆シミュレーション

5.2　2 次元配列による有限状相の表現

P_2 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，$Q_i(i=1,2,3,4)$ をそれぞれ $1,2,4,8$ 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`$[x\pm1]$`[`$y\pm1$`]` というように直接的に表せる（復号任意）．無限のセルをもつ状相を `c` で直接的に表すことはできない．我々は 1D-RPCA の有限状相をスタックで表現した方法 [5] を応用し，2D-RPCA 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

P_2 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，上記の矩形領域の定め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になるからである．

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．こ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^{-1} を表す計算で非可逆的に実現できる．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

P_2 の可逆シミュレーションは図 4 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 `gmap` を `t_end` 回適用して更新する．

図 5 に大域関数を計算する `gmap` を示す．まず遷移後の状相の表現を一時保存用の 2 次元配列 `c2` を割り付ける．次に図 6 の `table_expand` で遷移前の状相の表現をもつ `c1` の上下左右の境界を 1 セルずつ拡張する．この拡張には `local` と `delocal` を独立に使用する必要がある．したがって既存の Janus では実現できない．続く繰返しでは各セル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 `lmap` で局所関数を計算した結果をそれぞれ `c2` に格納する．`swap` は `c2` と全要素が 0 クリアされた `c1` の全要素を入れ替える．

図 7 に状相を表す 2 次元配列の遷移の例を示す．遷移ご

```
procedure gmap(int c1[][], int rule[][])
  local int c2[size(c1) + 2][size(c1) + 2] = {{0}},
      int u = 0, int l = 0, int r = 0, int d = 0
  call table_expand(c1)
  iterate int y = 0 to size(c2) - 1
    iterate int x = 0 to size(c2) - 1
      call get_ulrd(u,l,r,d,c1,x,y)
      local int t = 0
        call lmap(t, u, l, r, d, rule)
        t <=> c2[x][y]
      delocal int t = 0
    end
  end
  call swap(c2, c1)  // zero clear c2
  delocal int c2[size(c1)][size(c1)] = {{0}}, int u =
      0, int l = 0, int r = 0, int d = 0
```

図 5 2D-RPCA P_2 の大域関数

```
procedure table_expand(int t[][])
  local int t2[size(t)+2][size(t)+2] = {{0}}
  call move(t, t2)   // zero clear t
  delocal int t[size(t2)-2][size(t2)-2] = {{0}}
  local int t[size(t2)][size(t2)] = {{0}}
  call swap(t2, t)   // zero clear t2
  delocal int t2[size(t)][size(t)] = {{0}}
```

図 6 配列の拡大

```
{{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 7 左から初期状態，遷移前，遷移後の 2 次元配列

とに 2 次元配列の行と列が 2 ずつ拡大する．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．

5.5 状相の表現の最適化

前節で提案した 2D-RPCA の実装方法では，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最適化を行う方法を提案する．

5.5.1 効率的な状相の表現方法

有限状相を表す 2 次元配列は，配列の先頭と末尾の行と列にそれぞれ非静止状態が含まれる場合，整形という．例えば，図 7 の初期状態の配列と遷移前の配列は同じ状相を表している．したがって，整形な配列に写す大域関数を適用すると遷移後の状相が同じになり，単射性が失われる．5.4 節では非整形の配列に写す大域関数を用いることで単射性を実現していた．本節では図 7 の初期状態の配列のように，整形な配列のみを用いて単射性を実現する．

5.5.2 効率的な表現への変換

本節では，5.4 節で述べた遷移ごとに状相の表現が必ず増大する問題の解決案を提示する．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を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 へのゴミ出力を伴わない遷移を行うプログラムが実現可能である．

6 おわりに

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，状相の表現の最適化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の実現方法を示した．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ごとにメモリ使用量が増えるものとなった．

参考文献

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

[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).