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

2014SE048 木村孝大　2014SE114 矢澤拓海

指導教員：横山哲郎

1　はじめに

本論文では，可逆プログラミング言語 Janus を用いて可逆セル・オートマトン (RCA) のクリーン可逆シミュレーションを提案する．過去の研究 [1] にて 1 次元 RCA の可逆シミュレーションは提案されている．しかし，状相の遷移の度に新たな静止状態が増えてしまい，状相をスタックで表現する上で冗長となってしまうという問題があった．本研究では，既存のプログラムに対し可読性や空間効率が良くなるようリファクタリングを行い，遷移の際に無駄な静止状態が増えないようクリーンにシミュレーションを実現し，可逆性を保証することとメモリ消費を節約することを目的とする．シミュレーションを実現する際には過去の論文 [1] と同じく Janus を用いる．期待される効果として，有界で効率の良いセル空間表現によって 1 次元 RCA を静止状態以外の要素がある範囲に抑えて実現できることと，1 次元 RCA を可逆的に証明出来ることが挙げられる．

2　関連研究

本章では，本研究に関連する研究について述べる．本章ではセル・オートマトンの基本概要や定義について述べ，3 章以降にて，実装するために必要な概念や定義などをより詳しく述べる．

2.1　オートマトン

オートマトンとは常に 1 つの内部状態を持っており外部から連続している情報が入力され，それによって内部状態が遷移し，何らかの情報を出力するシステムのことをオートマトンと呼ぶ．オートマトンの内，内部状態が有限個であるものを有限オートマトンと呼ぶ．

2.2　セル・オートマトンの概要

セル・オートマトン (cellular automaton, 以下 CA) とはセルと呼ばれる大量の有限オートマトンを規則正しく配置したものであり，時間とともにそれぞれのセルの状態が他のセルの影響を受けて変化していくシステムである．

2.3　セル・オートマトンの定義

CA は以下の定義で与えられる．

$$A = \left(\mathbb{Z}^k, Q, (n_1, \ldots, n_m), f, \#\right)$$

$\mathbb{Z}^k$ は k 次元ユークリッド空間中の整数座標を持つ点集合であり，この点にセルを配置する．セルが配置されている，この空間をセル空間と呼ぶ．Q は各セルが取り得る内部状態の空でない有限集合である．$(n_1, \ldots, n_m)$ は $\left(\mathbb{Z}^k\right)^m (m = 1, 2, \ldots)$ の要素である．これは近傍と呼び，セルの状態が遷移する際に参照するセルのことである．関数 $f : Q^m \to Q$ は各々のセルの状態を決める局所関数である．局所関数は空間内の全てのセルに対して同時に適用される．これによって空間全体への状態，つまり状相が変化する．このように状相から状相への遷移関数を大域関数と呼ぶ．$q_1, \ldots, q_m, q \in Q$ に対し関係 $f(q_1, \ldots, q_m) = q$ が成り立つときそれを遷移規則と呼ぶ．したがって，f は遷移規則の集合で記述できる．$\#$ は静止状態を表し，$f(\#, \ldots, \#) = \#$ をみたす．これは空白に相当し，指定されない CA も存在する．$\alpha : \mathbb{Z}^k \to Q$ であるような写像 α を集合 Q 上の k 次元の状相と呼ぶ．α は A の状相ともいう．したがって，$x \in \mathbb{Z}^k$ とするとき，$\alpha(x)$ は座標 x の位置にあるセルの状態を表す．集合 Q 上の k 次元状相すべての集合を $\mathrm{Conf}_k(Q)$ で表す．つまり $\mathrm{Conf}_k(Q) = \{\alpha \mid \alpha : \mathbb{Z}^k \to Q\}$ である．k は基本的に前後関係からわかるため省略する．静止状態が指定された CA には有限状相と無限状相の概念が存在し，集合 $\{x[] \mid x \in \mathbb{Z}^k \wedge \alpha(x) \neq \#\}$ が有限の場合を有限状相，そうでない場合を無限状相と呼ぶ．

2.4　セル・オートマトンにおける可逆性

セル・オートマトンにおける可逆性は，大域関数に対する制約により定義される．可逆セル・オートマトン (reversible cellular automaton, 以下 RCA) とは，大域関数が全単射であるような CA, すなわち現在の状相に対し，直前の状相がちょうど 1 つ存在するような CA のことを言う．大域関数が単射であるか否かを判定するアルゴリズムは CA が 1 次元である場合にしか存在せず，RCA を実装するためには何らかの便法を用いる必要がある．

3　分割セル・オートマトン

分割セル・オートマトンは各セルがいくつかの部分に分割された構造を持つ CA である．分割 CA は可逆 CA を設計することができる CA の一種である．

3.1　分割セル・オートマトンの定義

分割 CA は次の式によって定義される．なお，一般的な CA と同様の要素については説明を割愛する．

$$A = \left(\mathbb{Z}^k, (Q_1, \ldots, Q_m), (n_1, \ldots, n_m), f, (\#_1, \ldots, \#_m)\right)$$

Q_i は各セルの第 i 部分 $(i = 1, \ldots, m)$ が取り得る，内部状態が空でない有限集合である．これは，各セルが m 個に分割されることを示す．$(\#_1, \ldots, \#_m) \in (Q_1 \times \cdots \times Q_m)$ は $f(\#_1, \ldots, \#_m) = (\#_1, \ldots, \#_m)$ を

満たす静止状態である．便宜上，静止状態 $\#_1, \ldots, \#_m$ は m 個の部分状態を同一視し，$(\#, \ldots, \#)$ として表す．標準的な CA 同様に，静止状態は空白に相当し，それが指定されない CA も存在する．標準的な CA と同分割 CA は，局所関数が単射である場合は可逆となることが知られている [2].

3.2 1 次元 3 近傍分割セル・オートマトン

分割 CA の具体例として過去の研究 [1] でプログラムの実現の際に利用された以下のような 1 次元 3 近傍分割 CA を挙げる．

$$A_{\mathrm{P}} = (\mathbb{Z}, (L, C, R), (1, 0, -1), f_{\mathrm{P}}, (0, 0, 0)) \quad (1)$$

$$L = C = R = \{0, 1\} \quad (2)$$

各セルは右部分，中央部分，左部分のように 3 つに分割されている．遷移の際には右隣のセル左の部分，着目セルの中央部分，左隣のセルの右部分の状態に依存する．分割されたセルの左部分を l, 中央部分を c, 右部分を r, と置くと遷移規則 $f(l, c, r) = (l', c', r')$ は 1 のように表現できる．1 次元 3 近傍 CA の局所関数が単射な例が表 1 であり，これを f_{P} とする．状態 1 を黒丸，状態 0 を空白とした時の A_{P} の挙動は図 1 のようになる．分割された各セルの右部分にある黒丸は右へ，左部分にある黒丸は左へと動くことが見て取れる．

表 1 単射な 1 次元 3 近傍分割 CA の遷移規則の例

r	c	l	l'	c'	r'
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	0	1	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	1
1	1	0	0	1	1
1	1	1	1	1	1

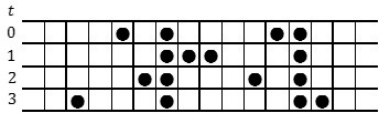

図 1 1 次元 3 近傍分割 CA の挙動例

4 クリーン可逆シミュレーション

Janus で RCA のクリーン可逆シミュレーションを実現するには，単射な大域関数 F をクリーンに実現する必要がある．RCA のシミュレーションを行う上で，状相の表現を行う上で無関係な情報が残らないならばクリーン可逆シミュレーションである．文献 [1] では単射な局所関数 f_{P} を用いた単射な大域関数 F_{P} を Janus でクリーンに実現することでオーバーヘッドを削減し，クリーン可逆シミュレーションが実現された．Janus プログラムの実行は，Janus のオンラインインタプリタ [3] を用いる

4.1 スタックを用いたセル空間の実現

RCA をシミュレートするには無限長のセル空間を有界なメモリ上で実現する必要がある．本節では RCA を有限状相のものに限定し，スタックを用いることでセル空間の実現を行う．静止状態のセルを除いたセルの個数が有限個である RCA では，静止状態のセルが無限に連続する．そこで，スタックの先頭と底には静止状態のセルが無限に連続しているものとしてその部分を省略し，残りの部分をスタックに格納する．これにより有限個の要素数で無限長のセル空間を表現し，有界なメモリ上での実現が可能となる．この方法では，静止状態のセルを除いたセルの個数が無限個である RCA，静止状態が 2 つ以上設定されている RCA を実現することができない．1 次元 3 近傍分割セル・オートマトンの場合は分割位置の明示化のため，スタックの先頭を分割されたセルの左部分とする．例えば，時刻 t において，RCA の各セルの状態を表す要素として 1,0 が存在し，0 は静止状態である．図 2 のような状相が存在するとすると，この状相はスタック sr を用いて

$$sr = 0 :: 0 :: 1 :: 1 :: 0 :: 1 :: 0 :: 1 :: [\,]$$

と表される．ここで，図 2 の状相に対し，以下の 2 種類の表現が可能であるとする．

$$sr = 0 :: 0 :: 1 :: 1 :: 0 :: 1 :: 0 :: 1 :: [\,]$$
$$sr = 0 :: 0 :: 1 :: 1 :: 0 :: 1 :: 0 :: 1 :: 0 :: [\,]$$

これらはどちらも同じ状相を表しているため，単射な大域関数を用いていたとすると，次の時刻に同じ状相へと遷移し，単射性が失われてしまう．文献 [1] ではこの問題を避けるため，これらを別のセル空間として扱い，同じセル空間に遷移しないようにして単射性を保っていた．しかし，この方法ではスタックの先頭と底に無駄な静止状態が増えてしまい，クリーンとは言えない．そこで我々は，セル空間の表現を分割位置が変更されず，かつスタックの底に静止状態が入らないように静止状態の数が最適化されたものに統一することで単射性を保ち，クリーンと言えるように実現した．

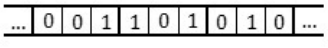

図 2 状相の例

4.2 既存の研究について

既存のプログラムのリファクタリングにあたりセルの遷移を表現するプロシージャについて，3 つの手法を提案した．主にその部分について述べるため既存の研究と大きく

変わらない部分については省略する．なお既存のプログラム同様，シミュレーションには 3.2 節で定義された RCA A_{P} を用いる．Janus プログラムの実行には Janus オンラインインタプリタ [3] を用いる．プロシージャ `calc` が，`rule` と `pop_lcr` で取り出した l, c, r の値を用いてセル単位での遷移を行う．このプロシージャのリファクタリング方法として 3 種類の方法が挙げられる．

4.3 セル単位での遷移方法

プロシージャ `calc` の遷移方法について記述する．我々はセル単位での遷移に対し，以下の 3 種類の手法を提案し，吟味を行なった．

- 遷移前のセルの状態が格納された二次元配列と遷移後のセルの状態が格納された二次元配列を用意し，遷移前の配列を走査することで添え字を計算し，遷移後の配列の同じ添え字の要素を使って遷移を行う方法
- 遷移後のセルの状態が格納された二次元配列を使用し，l, c, r の値から対応する遷移規則の添え字を計算することで遷移を行う方法
- 遷移後のセルの状態が格納された二次元配列を使用し，l, c, r の値によって条件分岐を行うことで遷移規則を選択する方法

本項ではそれらの手法に対し，順番に手法 A, 手法 B, 手法 C と称することとする．ここで，セルの分割数を n，分割されたセルのそれぞれの部分がとりえる内部状態の数を m とし，時間計算量の計算対象は条件式とアサーションでの比較の回数とする．また，どの遷移規則が出現する確率も同様に確からしいとする．空間計算量はプロシージャ `calc` において使用する変数の個数を基準とする．

4.3.1 手法 A

手法 A では，遷移前のセルの状態が格納された二次元配列 `pre_rule` と遷移後のセルの状態が格納された二次元配列 `post_rule` の 2 つの配列を用いる．まず変数 `l,c,r` の値を用いて配列 `pre_rule` を走査し，対応する添え字の値を計算し，int 型変数 `cnt` に格納する．その後，配列 `pre_rule` と変数 `cnt` を用いて変数 `l,c,r` の値をゼロクリアし，配列 `post_rule` と変数 `cnt` を用いて遷移後のセルの状態を計算し，変数 `l,c,r` に格納する．最後に，変数 `l,c,r` の値を用いて配列 `post_rule` を走査し，変数 `cnt` の値をゼロクリアする．空間計算量は $6m^n + 4$ となる．時間計算量は最低で $2n + 2$，最大で $2\,(n+1)\,m^n$，平均は $\left\{(1+m^n) \times \frac{1}{2}\right\} \times (n+1) \times 2$ すなわち $(m^n+1)\,(n+1)$ となる．内部状態の数やセルの分割数が増えても変更すべき箇所が少なく，拡張性は高い．プログラムは短く可読性は高いが，他の方法と比べて空間計算量も時間計算量も最も大きい．

4.3.2 手法 B

手法 B では，遷移後のセルの状態が格納された二次元配列 `rule` を用いる．このとき，配列 `rule` の添え字が遷移前のセルにおける `l,c,r` の値をこの順番で n 進数として見た場合の値となるようにする．変数 `l,c,r` の値を用いて対応する添え字の値を計算し，int 型変数 `rule_no` に格納する．その後，変数 `rule_no` 用いて変数 `l,c,r` の値をゼロクリアし，配列 `rule` と変数 `rule_no` を用いて遷移後のセルの状態を計算し，変数 `l,c,r` に格納する．最後に，変数 `l,c,r` の値を用いて配列 `post_rule` を走査し，変数 `cnt` の値をゼロクリアする．配列が 1 つであるので，空間計算量は A のおおよそ半分である $3m^n + 4$ となる．また，時間計算量は，最低で $n+1$，最大で $(n+1)\,m^n$，平均は $\frac{1}{2}\,(m^n+1)\,(n+1)$ である．プログラムも比較的短い．セルの分割数が増えた際の変更箇所は手法 A と同程度だが，内部状態の数が増えた場合は，添え字の計算方法を変更する必要がある．

4.3.3 手法 C

手法 C では，遷移後のセルの状態が格納された二次元配列 `rule` を用いる．まず変数 `l` の値を判定し，条件分岐を行う．次に変数 `c` の値を判定し，条件分岐を行う．その次に変数 `r` の値を判定し，条件分岐を行う．これにより，適用する遷移規則がわかるので，変数 `l,c,r` の値を定数でゼロクリアし，適切な配列 `rule` の要素を用いて遷移後のセルの状態を変数 `l,c,r` に格納する．アサーションは遷移後のセルの状態が何であるかによって判定を行う．空間計算量は B とほぼ等しく，$3m^n + 3$ となる．時間計算量は，最低で $\log 2\,(m^n) + n\,(m^n - 1)$ であり，最大も平均も等しい．遷移規則に応じて最適化が可能であり，計算量は少なくすることが可能．一般性を残すとプログラムが非常に長くなり，可読性が低いことが欠点である．また，状態数が変化した場合でもプログラムを大きく変更する必要があり，拡張性は低い．これらの解析により，我々は A_{P} 以外の分割 CA にも適用しやすく，かつ空間計算量と時間計算量の効率の良い手法 B を選択した．

表 2 `mypush` における入出力の対応

入力	出力
$s = [\,] \wedge x = \#$	$s = [\,] \wedge x = 0$
$s = [\,] \wedge x = v \neq \#$	$s = v :: [\,] \wedge v \neq \# \wedge x = 0$
$s = n \neq [\,] \wedge x = w$	$s = w :: n \wedge x = 0$

4.4 改善したプログラム

既存のプログラムでは空スタックに静止状態を push する際に制限をかけておらず，また，遷移を余分に 2 回行っており，結果的に無駄な静止状態が増えてしまっていた．この問題を解決するため，スタックに無駄な静止状態が入らないよう変更する必要がある．このとき，プログラムの

単射性が失われないよう，セル空間の表現をスタック内の静止状態の数が常に最適化されるものに統一しなくてはならない．そこで我々は，空スタックに静止状態を push する際にスタックの底に静止状態が入らないよう push を拡張し，mypush というプロシージャとして実装することで問題の解決を図った．プロシージャ mypush はスタック s と静止状態を含む何らかの値が入った int 型変数 x を入力とし，変数 x の値が先頭に入ったスタック s と中身が空になった変数 x を出力する．このとき，スタック s が空であり，かつ変数 x が静止状態と等しい場合，スタック s に対する操作は行われず，変数 x の値を空にする．プロシージャ mypush における入力と出力の関係は 4.3.3 のようになり，これらは背反であるため，可逆性が保たれる [4].

```
1  procedure mypush(int x, stack s)
2    //0 means blank
3    if empty(s) && x = 0
4        then x ^= 0
5        else push(x, s)
6    fi empty(s)
```

ここで，プロシージャ mypush 内の 0 は静止状態を表す．静止状態が 0 でない場合は，適宜変更が必要である．また，プロシージャ mypush を逆実行すると，空スタックからの pop を行った際に pop を行わず，代わりに変数に静止状態を入れるよう拡張したプロシージャ pop として実行できる．これらを用いて，プログラム内の pop,push をそれぞれ uncall mypush，call mypush と書き換えることでスタック内の静止状態の数の最適化が可能となる．この拡張により，提案プログラムは以下のような変更が加えられている．プロシージャ move_memory_start において，セル空間の表現をスタック内の静止状態の数が最適化されたものに統一したことによりセル空間の唯一性を保つ必要がなくなったため，セル空間の両端に静止状態を追加する部分を削除した．その代わりにセルの分割位置の情報を保存し，なおかつループの条件式とアサーションを成り立たせるために，スタック sl の要素を 3 つまとめて，つまりはセル単位でスタック sr に移動させるよう変更した．プロシージャ pop_lcr を使用する際，プロシージャ local_map 内の loop の条件式が複雑になると判断したため，pop_lcr を使用せず，ループ内で直接 l,c,r を pop することにし，loop の条件として追加した．プロシージャ pop_lcr を使用し，l,c,r をまとめて pop する場合，例えばセル空間が $\mathtt{sr} = 0 :: 0 :: 1 :: 0 :: 1 :: 1 :: []$ のように表現されている場合，変数 l,c,r が全て 0,0,0 となる．スタック sl へ遷移結果を push する際にプロシージャ mypush を使用しているため，sl には何も入らず，プロシージャ local_map 内のアサーションが成り立たなくなる．プロシージャ pop_lcr を使用しないことにより，プロシージャ init_config でスタック sr に静止状態を入れる必要がなくなったため，該当部分を削除した．また，既存のプログラムでは初期状態として与えるセル空間の表現に制限は無かったが，提案プログラムにおいて初期状態として与えることができるセル空間の表現は，スタックの要素数が最適化されたものに限定される．以下に 3.2 節の図 1 と同様に時刻 $t = 0$ に $sr = 0 :: 0 :: 1 :: 0 :: 1 :: 0 :: 0 :: 0 :: 1 :: 1 :: []$ と表される状相を与えた際のプログラムの実行例を示す．

```
1  sl = nil, sr = <0, 0, 1, 0, 1, 0,
       0, 0, 0, 1, 1]
2  sl = nil, sr = <0, 1, 1, 1, 0, 0,
       0, 1]
3  sl = nil, sr = <1, 1, 0, 0, 0, 1,
       0, 1]
4  sl = nil, sr = <1, 0, 0, 0, 1, 0,
       0, 0, 0, 0, 1, 1]
```

これらの変更により，静止状態が無駄に増えていく問題を解決し，クリーン可逆シミュレーションが実現できた．

5 おわりに

本研究では RCA を Janus プログラムとして直接シミュレーションする既存の研究に対し，リファクタリングを行なった．その結果，既存の研究では実行結果が時間に線形に比例して大きくなっていたのに対し，静止状態以外の要素の範囲に線形に抑えることに成功した．これにより，無限長のセル空間を持つ 1 次元 RCA を，有界な実行結果で，クリーン可逆シミュレーションを実現した．そして，可逆性が保証されているプログラミング言語 Janus にてプログラムを実装したことによって，1 次元 RCA に可逆性を与えられたことを証明できたと言える．今後の課題としては 2 つの方面での RCA の拡張が挙げられる．今回実装した 1 次元 RCA において，様々な遷移規則に対して，より遷移規則を与えやすい遷移手法を提案，ならびに評価することである．もう一つは，今回得られた知見を駆使することによって多次元 CA の可逆シミュレーション，可逆性を与えられることの証明を行うことである．

参考文献

[1] 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．南山大学情報理工学部 2013 年度卒業論文 (2014)

[2] 森田憲一：可逆計算，ナチュラルコンピューティング・シリーズ Vol. 5, 近代科学社 (2012)

[3] Janus Playground, available from http://tetsuo.jp/janus-playground/

[4] Yokoyama, T., Axelsen, H.B. and Glück, R. : Fundamentals of reversible flowchart languages, Theoretical Computer Science, Vol.611, pp.87–115 (2016)

[5] Yokoyama, T., Axelsen, H.B. and Glück, R.: Principles of a reversible programming language, Proc. Computing frontiers (CF'08), pp.43–54 (2008).