書き換えルールによるノノグラムの解法の表現

札幌学院大学　社会情報学部社会情報学科　小　池　英　勝

要旨：著者は，多様な計算を許す並列処理において，効率と信頼性を両立するための研究を行ってきた．その例題の１つとしてノノグラムがある．ノノグラムの問題は，粗粒度並列性を持ち，縦横の升目の並びは，それと交差して升目を共有する他の並びと互いに従属関係を持つ．ノノグラムは NP 完全に分類されるため，プロセスが単純に総当たり方式で問題を解こうとすると，すぐに扱えるパズルサイズに限界が訪れる．一方，総当たり方式の替わりに，人間がパズルを解く際のヒューリスティックスを用いれば，効率は大きく改善される．本論文では，ノノグラムの並列処理において分割されたサブ問題を処理する個々の逐次プロセスの高速化に取り組む．人間がパズルを解く際に用いるノノグラムの解法の定石を書き換えルールで表現することで逐次プロセス構築し，その効率を評価する．そして，そのルールを用いて効率的な並列プログラムの自動生成に向けた目標設定を行う．

キーワード：等価変換計算モデル，ノノグラム，効率化，言語処理系

1　はじめに

これまでに，等価変換計算モデル（Akama and Nantajeewarawat, 2006）と等価変換を正しく並列実行する枠組み（Akama et al., 2006）に基づく，制約充足問題を並列処理するための研究（Koike and Akama, 2011）の中で，ノノグラムをその一例として扱ってきた．並列計算の処理効率は，各プロセスのインタラクションの効率と同時に，各プロセスの効率が重要であることは明らかである．これまでの実装では，文献(小池，2011)にあるように，個々のプロセスで分割されたサブ問題の解をすべて集め，その共通する部分を抽出して，分割前のもとの問題に反映させることで，計算の正当性を保証していた．この方法は，プログラムの全ての処理の正当性が比較的容易に証明できるという大きな利点があった．プログラムは，等価変換計算モデルに基づく言語処理系 ETI（Koike et al., 2005）で動作するルールベースの抽象プログラムとして表現され実行することが可能である．更に，抽象プログラムを C＋＋プログラムに変換して，定数倍の処理効率の改善を行うことができる．ここで，効率改善のために重要なのは，抽象プログラムのレベルでいかに高度な効率化を行うかである．抽象プログラムのレベルでは計算量のオーダーが変わる効率の改善の可能性があるが，C＋＋プログラム等に変換する低レベルの効率化は一般に定数倍の改善しか見込めない．ノノグラムを解く問題は NP 完全であるためそのサイズが大きくなると，上述の解をすべて集める方法では処理時間が長すぎるという問題が起こる．一方人間がノノグラムを解く場合，ある程度習熟してくると，上述の方法では解けないような問題でも，数時間

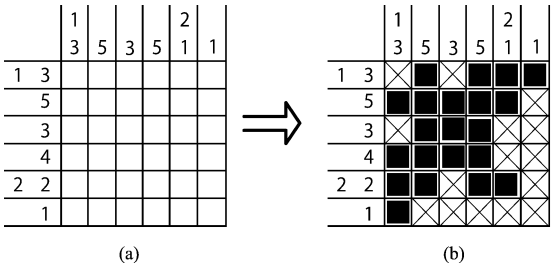

図 1 ：ノノグラムの問題の例

から数日で解くことができる．人間がどのようにノノグラムを解くかは (Wikipedia, 2012) で紹介されている．本論文では，効率的なプログラムを自動生成するという目的のために，まず，生成するべきプログラムの特定を行う．具体的には，上述のノノグラムの人間の解法を，ルールベースの抽象プログラムで記述し再現する．

以降の本論文の構成を示す．2 節で，ノノグラムについて述べる．3 節で等価変換計算モデルに基づくプログラム生成について説明する．4 節で，ノノグラムの人間の解法について説明し，それをルール集合（抽象プログラム）として表現する．5 節で，実験結果を示す．6 節で，本論文のまとめを述べる．

2　ノノグラムのルール

ノノグラム（Nonogram）とは，お絵かきロジック，ピクロス，お絵かきパズルなどの別名を持つパズルである．このパズルの目的は，N×M の空白の升目の横と縦に配置された数字をヒントに黒く塗り潰すか空白のままにするかを決定し絵を完成させることである．ただし，N と M は自然数である．横と縦に配置された数字は，連続して塗り潰す枡目の数を表している．ただし，2 個以上の数字が与えられている場合は，連続して塗り潰す領域の間に 1 個以上の空白の升目を存在させなければならない．図 1 (*a*) はノノグラムの問題の例である．図 1 (*b*) は (*a*) を解いた状態である．塗り潰す升目は■で，空白が確定した升目は×で表されている．

3　等価変換計算モデルに基づく解法

3.1　形式的仕様

本節で，ノノグラムの形式的仕様を問題記述として定義する．ノノグラムの問題記述の定義部 D は図 2 のような節集合である．この定義では，塗り潰すべき升目を 1 で，空白にするべき升目を 0 で，どちらか判明していない状態を変数で表している．本論文では，変数は＊で始まる記号，又は，？で表す．特に，？は無名変数を表す．最初の 3 つの節で述語 pat/2を定義している．pat/2述語は，ノノグラムのルールを表している．第 1 引数は，塗りつぶす枡目の制約を表す整数のリストで，第 2 引数は，升目を表すリストである．1 番目の節は，制約の数字が無い場合は第 2 引数のリストの全ての要素が白い枡目 (0) になることを定義している．2 番目と 3 番目の節は，制約の数字が 1 つ以上指定されている場合の定義である．2 番目の節は，最左の升目が白い升目の場合の定義である．3 番目の節は，最左の升目が塗り潰された場合の定義であり，この場合，第 1 引数の最初の要素で指定された数だけ，黒い升目 (1) が連続すること，更にその後に白い升目 (0) が続くことを，seq1/3述語と start0/1述語でそれぞれ定義している．4 番目と 5 番目の節で述語 seq1/3を定義している．これらの節で，seq1/3は第 2 引数のリストは始めの要素から，第 1 引数で与えられた整数 n の数だけ 1 が続き，第 3 引数は，第 2 引数のリストの $n+1$ 番目以降の要素で構成されるリストであることを定義している．6 番目と 7 番目の節で述語 start0/1を定義している．これらの節で start0/1は第 1 引数が空リストか，または，要素が 0 で始ま

```
(pat () *pl)<--(allZero *pl).
(pat (*n|*ns) (0 | *pls))<--
     (check_const (*n|*ns) *pls), (pat (*n|*ns) *pls).
(pat (*n|*ns) (1 | *pls))<--
    (subtract *n 1 *n2),(seq1 *n2 *pls *rest),
    (start0 *rest), (pat *ns *rest).
 (seq1 *n (1|*pls) *pls2)
    <--(> *n 0), (:= *n2 (- *n 1)),
       (seq1 *n2 *pls, *pls2).
(seq1 0 *pls *pls2)<--(= *pls *pls2).
(start0 ())<--.
(start0 (0|*rest))<--.
(allZero ())<--.
(allZero (0 | *rest))<--(allZero *rest).
(check_const *ns *pl)<--
    (length *ns *nl), (length *pl *pll),
    (listSum *ns *nsum), (subtract *nl 1 *nl2),
    (add *nsum *nl2 *ml), (ge *pll *ml).
```

図 2 ：ノノグラムの問題記述の定義部 D

```
(ans
    (*11 *12 *13 *14 *15 *16
     *21 *22 *23 *24 *25 *26
     *31 *32 *33 *34 *35 *36
     *41 *42 *43 *44 *45 *46
     *51 *52 *53 *54 *55 *56
     *61 *62 *63 *64 *65 *66)) <--
    (pat (1 3)  (*11 *12 *13 *14 *15 *16)),
    (pat   (5)  (*21 *22 *23 *24 *25 *26)),
    (pat   (3)  (*31 *32 *33 *34 *35 *36)),
    (pat   (4)  (*41 *42 *43 *44 *45 *46)),
    (pat (2 2)  (*51 *52 *53 *54 *55 *56)),
    (pat   (1)  (*61 *62 *63 *64 *65 *66)),
    (pat (1 3)  (*11 *21 *31 *41 *51 *61)),
    (pat   (5)  (*12 *22 *32 *42 *52 *62)),
    (pat   (3)  (*13 *23 *33 *43 *53 *63)),
    (pat   (5)  (*14 *24 *34 *44 *54 *64)),
    (pat (2 1)  (*15 *25 *35 *45 *55 *65)),
    (pat   (1)  (*16 *26 *36 *46 *56 *66)).
```

図 3 ：質問部 Q

る任意のリストであることを定義している．8 番目と 9 番目の節で allZero/1を定義している．これらの節で allZero/1は第 1 引数が，空リストかまたは全ての要素が 0 のリストであることを定義している．10番目の節は，check_const/2述語を定義している．この述語は，第 1 引数に縦または横の制約を表す数字のリストを，そして，第 2 引数に升目を表すリストを取る．この述語は，ノノグラムのルールに従って，制約のリストを満たすのに十分な枡目の個数があることを定義している．実際には，更にこの節のボディに現れる述語の定義が必要であるが，その述語の意味を説明するにとどめ，ここでは省略する．length/2は第 1 引数のリストの要素の個数が第 2 引数であること表す．listSum/2は第 1 引数のリストの要素の合計が第 2 引数であることを表す．subtract/3は第 1 引数と第 2 引数の差が第 3 引数である関係を表す．add/3は第 1 引数と第 2 引数の和が第 3 引数である関係を表す．ge/2は第 1 引数が第 2 引数以上である関係を表す．

質問部 Q は図 3 で示される 1 つの節で構成される節集合である．Q の節は，図 1 に示された解くべき 6×6

枡の問題を表している．ヘッドの引数のリストは，ノノグラムの升目を表す．1 番目から 6 番目までの pat アトムは，ノノグラムの横の制約を表す．7 番目から12番目までの pat アトムは，ノノグラムの縦の制約を表す．それぞれの pat アトムは他の交差する 6 つのアトムと変数を 1 つずつ共有している．

解くべき問題の記述 S は $S=D\cup Q$ である．等価変換計算モデルでは，問題記述 S は宣言的な記述であり手続き的な意味は持たない．問題記述 S は，解くべき問題の宣言的意味 $\mathcal{M}(S)$ を定義する．

3.2 プログラム生成

等価変換計算モデルに基づく問題記述からのプログラム生成方法が提案されている（Koike et al., 2001; Akama et al., 2007）．これらの生成方法に従い，問題記述述 S から図 4 に示されるプログラムが生成される．このプログラムの具体的な生成方法と手続き的な意味の解説は，文献（小池，2011）にある．図 4 のプログラ

```
?-(RuleClassOrder 0 1),(RuleClass pat2 1),
  (RuleClass otherwise 0).

(pat (*n|*ns) ())==>{(false)}.
(pat () *PL)==>(allZero *PL).

pat2
(pat (*n|*ns) (*p|*pls))
    ==>{(= *p 0)},(check_const (*n|*ns) *pls),
                  (pat (*n|*ns) *pls);
    ==>{(= *p 1),(:= *n2 (- *n 1))},
                  (seq1 *n2 *pls *rest),
                  (start0 *rest),
                  (pat *ns *rest).
check_const
(check_const *n *pl),
  {(length *n *nl),(length *pl *pll),
   (listSum *n *ns),(:= *nl2 (- *nl 1)),
   (:= *ns2 (+ *ns *nl2))}
==>{(>= *pll *ns2)}.

(seq1 *N *PL *PL2),{(> *N 0)}
    ==>{(= *PL (1 | *PLS)), (:= *N2 (- *N 1))},
       (seq1 *N2 *PLS *PL2).
(seq1 0 *PL *PL2)==>{(= *PL *PL2)}.

(start0 ())==>.
(start0 (*p | *rest))==>{(= *p 0)}.

(allZero ())==>.
(allZero (*p | *rest))
    ==>{(= *p 0)}, (allZero *rest).

(length () *l)-->(= *l 0).
(length (*a | *rest) *l)-->(length *rest *l1),
                            (:= *l (+ *l1 1)).

(listSum () *s)-->(= *s 0).
(listSum (*n | *rest) *s)
    -->(listSum *rest *s1),(:= *s (+ *n *s1)).
```

図 4 ：自動生成されたプログラム Prog2

ムの名前を原文からそのまま引き継ぎ Prog2 とする．

3.3 プログラムの実行

問題記述 S の Q を Prog2 を用いて変換を繰り返すと，以下の節集合 Q' を得る．

```
(ans (0 1 0 1 1 1 1 1 1 1 1 0 0 1 1 1 0 0
      1 1 1 1 0 0 1 1 0 1 1 0 1 0 0 0 0 0))<-.
```

この計算結果は，1つの解を得たと同時にそれ以外の解が存在していないことも保証している．また，宣言的意味は保存されているので，$\mathcal{M}(S)=\mathcal{M}(D\cup Q')$ である．

4 人による解法の手順の再現

ノノグラムの解法には，ある条件で升目の値を決める定石がいくつかある．それらを書き換えルールとして記述する．その際に文献（Wikipedia, 2012）に記述されている各解法をプログラムとして記述し自動実行可能にした．プログラムは，等価変換計算モデルに基づく言語処理系 ETI で実行可能な書き換えルールとして記述した．書き換えルールは，頭部，適用条件部，置き換え部からなり，書き換え対象の節に対して，頭部がその節のボディアトムにマッチして，かつ，適用条件が満たされるならば，マッチしたアトムを書き換え部に置き換える．このルールの文法でノノグラムの解法を表す各処理を記述した．書き換えルールの一般的な説明は（小池，2011）にある．以降の小節で，ノノグラムの解法と書き換えルールとの関係を示す．

4.1 人による解法

文献（Wikipedia, 2012）によれば，ノノグラムの解法は3段階に分けられる(1)．第1段階の処理は，パズルを解く作業の初期に行われ，制約の数列から升目の値が確定する場合の処理である．第2段階は，第1段階，もしくは，第3段階によって，値が確定された升目が現れた場合に，その情報と，制約の数列を合わせることで，新たな升目の値を特定する処理である．第3段階は，第1，第2段階いずれも新しい升目の値が確定できなくなった場合に，値が確定していない升目を任意に選び，塗りつぶさない場合と，塗りつぶす場合に分けて計算を進める処理である．

4.2 第1段階

4.2.1 0の処理

制約の数列が0（又は与えられていない）の場合は，その升目の並び全体を白い升目と確定できる．この状況を表すアトムの一例は，

```
(pat () (? ? ? ? ? ? ? ? ? ?))
```

であり，このアトムは

```
(pat () (0 0 0 0 0 0 0 0 0 0))
```

と変換できる．この処理を書き換えルールで表すと，

```
(pat () *PL)==>(allZero *PL).
```

となる．このルールは Prog2 に含まれているものと同一であるので，自動生成可能である．

4.2.2 最高値の処理

与えられた制約の数列によって，塗りつぶしの仕方が一意に定まる場合の処理である．最高値とは，パズルの縦又は横の升目の個数である．この処理が行える条件が満たされるときは，制約の数列の合計＋数字の個数－１が最高値と等しくなる．Wikipedia の記述では，制約の数列の要素の個数が１つのときとそれ以上のときで場合分けされているが，書き換えルールで表現すると，１つ以上の場合としてひとまとめに出来る．この状況を表すアトムの一例は，

```
(pat (7 2) (? ? ? ? ? ? ? ? ? ?))
```

であり，このアトムは，

```
(pat (7 2) (1 1 1 1 1 1 1 0 1 1))
```

と変換できる．この処理を書き換えルールで表すと，

```
(pat (*n | *N) (*p| *pl)),
    {(/== *p 0),(length (*p| *pl) *pn), (listSum (*n | *N) *sn),
     (length (*n | *N) *nn), (:= *pn (- (+ *sn *nn) 1))}
    ==>{(fillOneZero (*n|*N) (*p| *pl))}.
```

となる．fillOneZero/2は D アトムであり，第１引数の数列に従って，第２引数のリストが塗りつぶされていることを表す．

4.2.3 左右につめた時に生じる共通の黒い升目の処理

この処理は，可能な解の中から最左のもとの最右のものを比較して，同じ数字から由来する塗りつぶしが同じ升目にある場合にその升目を黒い升目に確定するものである．この状況を表すアトムの一例は，

```
(pat (4 2) (? ? ? ? ? ? ? ? ? ?))
```

であり，このアトムは，

```
(pat (4 2) (? ? ? 1 ? ? ? ? ? ?))
```

と変換できる．この例では，４番目の升目だけが，最右の解

```
(pat (4 2) (1 1 1 1 0 1 1 0 0 0))
```

と最左の解

```
(pat (4 2) (0 0 0 1 1 1 1 0 1 1))
```

で共通しているため，1と確定することができる．この処理を書き換えルールで表すと，

```
(pat *N *pl),
    {(not (var *pl)), (LRCommon *N *pl *pl2),
     (copy *pl *plc),(= *plc *pl2),(moreSpecialized *pl *pl2)}
        ==>{(= *pl *pl2)},(pat *N *pl).
(pat *N *pl),
    {(not (var *pl)), (not (LRCommon *N *pl *pl2))}
        ==>{(false)}.
```

となる．LRCommon/3は，第1引数の数列と第2引数の升目のリストから作られる最左の解と最右の解の共通部分から確定する升目を発見し黒く塗りつぶした結果を第3引数として反映させる．このアトムを処理するルールは，黒い升目以外にも，外側の白い升目として確定する升目についても処理を行う．更に，このアトムを処理するルールは，升目の状態に矛盾が発生した場合失敗する．上記2番目のルールはこのことを利用して，無駄な計算の枝刈りを行う．

4.3 第2段階

4.3.1 全ての黒い升目が確定した処理

この処理は，言い換えると「未確定の升目を全て白い升目にする」処理である．与えられた制約の数列を全て満たすように升目が確定した場合，残りの升目は全て白い升目として確定する．この状況を表すアトムの一例は，

```
(pat (1 1) (? 1 ? ? ? ? ? 1 ? ?))
```

であり，このアトムは，

```
(pat (1 1) (0 1 0 0 0 0 0 1 0 0))
```

と変換できる．この処理を書き換えルールで表すと，

```
(pat *n *pl),
    {(listSum *n *ns), (listSumVar *pl *pln),
     (== *ns *pln)}==>(fillZero *pl).
```

となる．listSumVar/2は，第1引数として与えられたリストの要素の合計を第2引数とするDアトムである．

ただし，変数は 0 として扱う．fillZero/1は，引数として与えられたリストの未確定の升目を全て白い升目とする N アトムである．

4.3.2 全ての白い升目が確定した処理

この処理は，言い換えると「未確定の升目を全て黒い升目にする」処理である．未確定の升目の数が，制約の数列の合計と黒い升目として確定した升目の数の差と等しい場合，未確定の升目は，黒い升目として確定する．この状況を表すアトムの一例は，

```
(pat (1 1) (0 ? 0 0 0 0 0 1 0 0))
```

であり，このアトムは，

```
(pat (1 1) (0 1 0 0 0 0 0 1 0 0))
```

と変換できる．この処理を書き換えルールで表すと，

```
(pat *n *pl),
    {(listSum *n *ns), (listSumVar *pl *numOnes),
     (countVar *pl *vs), (:= *ns2 (- *ns *numOnes)),
     (== *ns2 *vs)}==>{(fillOne *n *pl)}.
```

となる．fillOne/2は，第 1 引数で与えられた制約の数列で表された条件を満たすように第 2 引数のリストの未確定の升目を塗りつぶす．

4.3.3 黒い升目の両隣を 0 で埋める処理

この処理は，制約の数列の最大値と同じ個数黒い升目が連続した場合，その直近の左右の升目が未確定ならば白い升目として確定する．この状況を表すアトムの一例は，

```
(pat (1 2 1) (? ? ? ? ? 1 1 ? ? ?))
```

であり，このアトムは，

```
(pat (1 2 1) (? ? ? ? 0 1 1 0 ? ?))
```

と変換できる．この処理を書き換えルールで表すと，

```
(pat *n *pl),
    {(copy *pl *pl2),(max *n *m),
     (findMaxBar *pl2 *m),(moreSpecialized *pl *pl2)}
        ==>{(= *pl *pl2)},(pat *n *pl).
```

となる．findMaxBar/2は，第２引数で与えられた数だけ連続する黒い升目の両端を白い枡目として確定する．

4.3.4 端の処理

端の升目が黒い升目と確定すると，それに対応する制約の数の分だけ連続して黒い升目が確定できるのでそれを処理する．この状況を表すアトムの一例は，

```
(pat (3 1) (1 ? ? ? ? ? ? ? ? ?))
```

であり，このアトムは，

```
(pat (3 1) (1 1 1 ? ? ? ? ? ? ?))
```

と変換できる．この処理を書き換えルールで表すと，

```
(pat (*n|*N) (1|*rest)),
    {(>= *n 1),(copy *rest *rest2),
     (:= *n1 (- *n 1)),(fillOne *n1 *rest2),
     (tail *n1 *rest2 *next),(start0 *next),
     (moreSpecialized *rest *rest2)}
        ==>{(= *rest *rest2)},(pat *N *next).
(pat *RN *RL),
    {(reverse *RN (*n|*N)), (reverse *RL (*re |*rest)),
     (== *re 1), (>= *n 1), (copy *rest *rest2),
     (:= *n1 (- *n 1)),(fillOne *n1 *rest2),
     (tail *n1 *rest2 *next),(start0 *next),
     (moreSpecialized *rest *rest2)}
        ==>{(= *rest *rest2),(reverse *N *RN2),
            (reverse *next *rnext)},(pat *RN2 *rnext).
```

となる．１番目のルールが左端の処理を行うルールである．２番目のルールは，制約の数列と，升目のリストを反転させて，１番目のルールと同様のことを行うことで右端の処理を行っている．

4.3.5 狭小マスの処理

白い升目で囲まれた連続する未確定の升目の個数が与えられた制約の数列の最小値よりも小さい場合，囲まれた未確定の升目を白い升目に確定する．この状況を表すアトムの一例は，

```
(pat (3) (? ? ? 0 ? ? 0 ? ? ?))
```

であり，このアトムは，

```
(pat (3) (? ? ? 0 0 0 0 ? ? ?))
```

と変換できる．この処理を書き換えルールで表すと，

```
(pat (*n|*N) *pl),
    {(findZero *pl *p),(> *n *p),(copy *pl *pl2),
     (fillZero *p *pl2), (moreSpecialized *pl *pl2)}
        ==>{(= *pl *pl2)},(pat (*n|*N) *pl).
(pat *RN *RL),
    {(reverse *RN (*n|*N)), (reverse *RL *pl),
     (findZero *pl *p),(> *n *p),(copy *pl *pl2),
     (fillZero *p *pl2), (moreSpecialized *pl *pl2)}
        ==>{(= *pl *pl2)},(pat *RN *RL).
(pat (*n) *pl),
    {(findVarsSectionEnclosedByZero *n *pl *start *end)}
        ==>{(fillZero *start *end *pl)}, (pat (*n) *pl).
```

となる．1番目のルールは，左端の狭小マスの処理である．2番目のルールは，右端の処理である．3番目のルールは，それ以外の狭小マスの処理である．findZero/2は，第1引数で与えられた升目のリストにおいて，左端から調べて最初に白い升目が現れる位置を第2引数とする．ただし，白い升目が現れる前に黒い升目が現れると失敗する．findVarsSectionEnclosedByZero/4は，第1引数で与えられた数より狭い間隔の白い升目で囲まれた未確定升目の位置を，第2引数のリストから探し，その始点と終点を第3引数と第4引数とそれぞれ対応させる．

4.3.6 確実に黒い升目が届く升目，届かない升目を処理

この処理は，左右につめた時に生じる共通の黒い升目の処理の書き換えルールで実現できる．

4.3.7 端や最高値の更新に対する処理

この処理は，個々の処理の実装で自然に実現されている．

4.4 第3段階

第3段階では，第1段階，第2段階だけでは，確定しない升目を処理する．この段階で未確定な升目は，与えられた制約の数列と現時点での確定した升目の情報だけでは，値を決定できないものである．このような升目は，黒い升目にするか，白い升目にするかで計算を分岐させる．この状況を表すアトムの一例は，

```
(pat (3) (? ? ? 0 0 0 0 ? ? ?))
```

であり，例えば，左端の升目を白い升目にしたものと黒い升目にしたものに計算を分岐させると，

```
(pat (3) (0 ? ? 0 0 0 0 ? ? ?))
```

と

```
(pat (3) (1 ? ? 0 0 0 0 ? ? ?))
```

に変換できる．変換後のアトムからさらに，それぞれ，

```
(pat (3) (0 0 0 0 0 0 0 1 1 1))
```

と

```
(pat (3) (1 1 1 0 0 0 0 0 0 0))
```

が得られる．実際の計算では，他のアトムと変数を共有しているので，他のアトムの変数が具体化されたときに一方から矛盾が導き出され，他方はそのまま残る．この処理は，Prog2 に含まれる分岐ルールで実現できる．よってこの処理を行う書き換えルールは自動的に生成できる．

5　実験と結果

5.1　アトム単位の計算

4 節で導入したルールを Prog2 に追加したプログラムを定石プログラム 1 と呼ぶことにする．表 1 のアトムを Prog2 と定石プログラム 1 でそれぞれ実行したときの処理時間を表 2 に示す．

表 1 について説明する．第 1 列は，アトムパターンの識別子で，表 2 で参照するために用いられる．第 2 列は，アトムパターンを表す．表 2 について説明する．第 1 列は，表 1 のアトムパターンの識別子である．第 2 列は，可能な解の個数である．第 3 列は，Prog2 で計算したときの所要時間で，単位は msec である．第 4 列は，定石プログラム 1 で計算したときの所要時間で，単位は msec である．

表 2 をみると，Prog2 では，可能な解の個数に応じて処理時間が大きく変動しているのに対して，定石プログラム 1 は，解の個数によらずその変動は少ない．そして，処理時間がどのパターンでも35msec 以下であり，Prog2 と比較して大幅に効率化されている．

表 1 ：実験に用いたアトムパターン

ID	アトムパターン
1	(pat (2 1 1 1 3 1 1 2 2 3) (? ?))
2	(pat (1 1 1 1) (? ?))
3	(pat (1 1 4 2 2) (? ?))
4	(pat (2 1 1 1) (? ?))
5	(pat (1 1 1 1 1)) (? ?))
6	(pat (1 1 1 1 1) (? ?))
7	(pat (1 1 1 1 1) (? ?))

表 2 ：アトムパターンの計算時間

ID	解の個数	Prog2［msec］	定石プログラム 1 ［msec］
1	1001	1313	35
2	17550	27880	32
3	4368	2204	25
4	220	305	35
5	20349	31169	25
6	4368	2818	19
7	462	230	14

表 3 ：パズルの処理時間

問題 ID	問題のサイズ	Prog3［msec］	定石プログラム 2 ［msec］
19914	25×25	13600	544937
19893	25×25	112412	4252627
19986	15×15	611	13849
19953	25×25	28950	1643105
17856	30×30	1993717	1103769

5.2 パズルの計算

表 3 は，ワールド機構を用いた効率化を行ったプログラム Prog3（小池，2011）による計算時間と Prog3 に節 4 で導入したルールを追加したプログラム（定石プログラム 2 ）の計算時間を表す．Prog3 は Prog2 の制御部分を効率化したバージョンであり，実行結果は Prog2 と同等である．実験に用いた問題は，お絵かきロジックのサイト[(2)] に掲載されているものから抽出したものである．実際の問題は，サイトの URL と表 3 中の問題 ID と .html を結合することで得られる URL を用いて閲覧することができる．実験環境は，CPU Intel Q96504 コア 3 GHz，メモリ 6 GB，OS 64ビット版 Windows7 Ultmate である．

表 3 について説明する．最初の列はノノグラム問題の識別子である．第 2 列は問題のサイズを表す．第 3 列は，プログラム Prog3 の実行時間で，単位は msec である．第 4 列は，定石プログラム 2 の計算時間である．この結果をみると，問題17856以外の処理時間が，定石ルールを導入したことによってむしろ長くなっている．この原因は，定石ルールの適用条件の判定が複雑なためその処理に時間が取られていることと，今回の定石ルールだけでは，Prog3 で行っている全ての通りを計算し尽くして結果を得る方法よりも計算過程の枝刈りの機会を逃しているためと考えられる．そのため，Prog3 では早期に排除される余分な計算を，定石プログラム 2 では行っている可能性がある．しかし，問題17856の結果の様にパズルのサイズが大きくなると，全通りの計算の不利が顕著になるため，定石プログラム 2 の方が処理時間が短くなっている．

6 おわりに

本論文では，制約充足問題の例としてノノグラムをとりあげ，その解法の定石を書き換えルール集合として表現した．書き換えルール集合は，等価変換計算モデルに基づく言語処理系 ETI で実行可能な抽象プログラムである．このプログラムを実行し，従来の全通りを計算し尽くす方法と比較した．オリジナルの解法の定石は，人間がパズルを解くことを想定して記述されている．そのため，それをプログラムとして表現すると，難易度

や複雑さは人間の感覚とは必ずしも一致しないことが分かった．オリジナルの解法の定石は，３段階に分けられ，第１段階，第２段階，第３段階の順で解くとされている．等価変換計算モデルに基づく解法の観点からは，第３段階は自動生成可能な分岐ルールに相当し，最も簡単に生成できるものである．また，これを用いれば原理的に全ての解を余すとことなく求めることができるし，それ以外に解が無いことも保証できる．しかし，このルールだけでは，１つの升目毎に常に正解と不正解（塗りつぶすか塗りつぶさないか）の計算パスを作り出すため効率が悪い．第１段階は，プログラムと実装してみると第２段階の特殊形である物があった．計算効率を上げるためには，第１段階，第２段階に相当するルールをいかに生成するかが問題になると分かった．特に，第１段階と第２段階を両方とも担当する処理のルールは，その複雑さから，人間にとっても，プログラム生成の観点からも難しいと考えられる．本論文では，第１段階と第２段階のルールは人手で記述したが，本研究の今後の目標は，これらのルールが形式的仕様に関して正当であることを証明し，そして，これらのルールと同等以上の性能を持ったルールを自動生成することである．本論文では，その目標を具体的に示した．

謝辞

本研究は2011年度札幌学院大学研究促進奨励金（SGU-S11-198023-01）の助成を受けた．

注

(1) ただし，この段階分けは必ずしも普遍的な解法に必要ではないと思われる．同項目の英語版の記述には段階分けの記述はない．

(2) http://www.minicgi.net/logic/

参考文献

Akama, K. and E. Nantajeewarawat (2006). “Formalization of the Equivalent Transformation Computation Model”. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, **10**(3) pp. 245–259.

Akama, K., E. Nantajeewarawat, and H. Koike (2007). “Program Generation in the Equivalent Transformation Computation Model Using the Squeeze Method”. In *PSI 2006*, volume 4378 of *Lecture Notes in Computer Science*, pp. 41–54. Springer-Verlag.

Akama, K., E. Nantajeewarawat, and H. Ogasawara (2006). “Generation of Correct Parallel Programs Based on Specializer Generation Transformations”. In *Proceedings of the 7th international conference on intelligent technologies (InTech'06)*, pp. 90–99.

Koike, H. and K. Akama (2011). “Generation of Correct Parallel Programs Guided by Rewriting Rules”. *In Proceedings of The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, **1** pp. 12–18.

Koike, H., K. Akama, and E. Boyd (2001). “Program Synthesis by Generating Equivalent Transformation Rules”. In *in Proceedings of the Second International Conference on Intelligent Technologies (InTech'01)*, pp. 250–259.

Koike, H., K. Akama, and H. Mabuchi (2005). “A Programming Language Interpreter System Based on Equivalent Transformation”. In *2005 IEEE 9th International Conference on Intelligent Engineering Systems (INES 2005)*, pp. 283–288.

Wikipedia (2012).「お絵かきロジック」. http://ja.wikipedia.org/wiki/.

小池英勝（2011）.「等価変換計算モデルに基づくノノグラムの解法」. 情報科学，**31** pp. 37–56.

Expression of Solution for Nonogram Puzzles by Rewriting Rules

by

Hidekatsu KOIKE

Abstract

The author has studied efficient and correct parallel computation allowing various processes. Computing nonogram puzzles in parallel is an example of such computation. Parallel computation of nonogram has coarse-grained parallelism. Calculation of a chunk of cells in a nonogram puzzle depends upon other chunks which intersect the chunk and share cells. We are not able to treat puzzles of large size only by brute-force computation since solving nonogram puzzles is an NP-complete problem. The efficiency of the solution may be improved by using heuristics imitating nonogram solution techniques for human. This paper treats improvement of each sequential process in parallel computation of nonogram by expressing solution techniques for human by rewriting rules and evaluating their efficiency. After that, programs to be generated in the future study are determined by using the rules.

Keywords: Equivalent transformation computation model, Nonogram puzzle, Optimization, and Language processing system.

*Faculty of Social Information, Sapporo Gakuin University, koike@sgu.ac.jp