数理モデル化と問題解決 23－4
（1999. 2. 18）

等価変換による数独パズルの解法

赤間 清　小池 英勝　宮本 衛市
北海道大学 大学院 システム情報工学専攻
札幌市北区北13条西8丁目. Tel. 011-706-6814
{akama,koike,miya}@complex.eng.hokudai.ac.jp

問題記述を等価的に簡単な形に変形することによって，多くの問題を解くことが できる．等価変換パラダイムは等価変換を用いた簡単化によって問題を解くための一般的な枠組を提供している．本稿では，等価変換による問題解決の方法を あるパズル (数独パズル) の解法に適用することを試みる．本方法によれば，等価変換ルールの導入，ドメイン変数の自然な採用などによって問題を解くプログラムを 逐次的に改善し，問題の正しい答えを得ることができる．

等価変換ルール　パズル　データ構造　正当性

Solving Su-doku Puzzles by Equivalent Transformation

Kiyoshi Akama　Hidekatsu Koike　Eiichi Miyamoto
Dept. of System and Information Engineering, Hokkaido University
North 13 West 8 Kita-ku Sapporo-shi. Tel. 011-706-6814
{akama,koike,miya}@complex.eng.hokudai.ac.jp

Many problems can be solved by transforming the problem description equivalently into a simpler form. The equivalent transformation (ET) paradigm provides us with a general framework for solving problems by simplification in terms of equivalent trnsformation. In this paper we apply the ET problem solving method to Su-doku puzzles. In this approach we can successively improve ET programs by introduction of new ET rules and adoption of domain variables, thus finally obtain correct solutions of given puzzles.

equivalent transformation rule,　puzzle,　data structure,　correctness

1 まえがき

問題記述を等価的に簡単な形に変形することによって，多くの問題を解くことが できる．等価変換パラダイムは この問題解決方法を 一般的に定式化したものである．本稿では，等価変換による問題解決の方法を あるパズル (数独パズル) に適用することを試みる．本方法によれば，問題を解くためのルールを少しずつ追加し，それによる簡単化の効果を観察しながら，プログラムを逐次的に改善することができる．また 簡単化を進めるために 自然な形で 新しいデータ構造を導入できる．

数独パズルを解くために，等価変換プログラミング言語 ETC [9] を用いる．等価変換プログラミング言語 [9] は，等価変換プログラミングを支援する言語であり，そのプログラムは，等価変換ルールの記述 (ルールの適用条件と，変換方法)，制御の記述 (各ルールへのクラス指定と クラスの優先順位指定) という 2 種類の記述からなる．

2 数独パズルとその定式化

2.1 数独パズル

数独パズル (su-doku puzzle) は，n^2 種類の数字が書かれた $n^2 \times n^2$ の盤面（$n = 3$ の場合，たとえば 図 1 の **問題 1**）が与えられ，次の条件を満たすように盤面の空欄に数字を当てはめるパズルである．

1. 縦に並ぶ n^2 個のマスのそれぞれに，1 から n^2 までの数字が 1 個ずつ入る．
2. 横に並ぶ n^2 個のマスのそれぞれに，1 から n^2 までの数字が 1 個ずつ入る．
3. 太線で囲まれた $n \times n$ の枠内の n^2 個のマスのそれぞれに，1 から n^2までの数字が 1 個ずつ入る．

人間が楽しむ場合 普通 $n = 3,4,5$ 程度の問題が多い．本論文では，$n = 3$ と $n = 5$ の場合の問題を 例題として用いる．本論文でプログラムや例などを記述する場合，4 章までは $n = 3$ を主に用いる．それから 一般の n の場合を 推測するのは 容易であろう．

2.2 数独パズルの節表現

数独パズルの制約は，2つに分けることができる．1つは，初期配置の条件を満たすこと，もう 1 つは，数独パズルの配置条件を満たすことである．それを次の節で表す[1]．

```
(解答 *array) ← (初期配置 *array),
                (数独制約 *array).
```

[1]本論文では，アトムは S 式記法で表す．

	6		2		4		5	
4	7			6			8	3
		5		7		1		
9			1		3			2
	1	2						9
6			7		9			8
		6		8		7		
1	4			9			2	5
	8		3		5		9	

図 1: 数独パズルの例題（問題 1）

たとえば，**問題 1** (図 1) では，「初期配置」述語は次のように定義される．? は 無名変数である．

```
(初期配置 *array) ←
        (= *array ((? 6 ? 2 ? 4 ? 5 ?)
                   (4 7 ? ? 6 ? ? 8 3)
                   (? ? 5 ? 7 ? 1 ? ?)
                   (9 ? ? 1 ? 3 ? ? 2)
                   (? 1 2 ? ? ? ? ? 9)
                   (6 ? ? 7 ? 9 ? ? 8)
                   (? ? 6 ? 8 ? 7 ? ?)
                   (1 4 ? ? 9 ? ? 2 5)
                   (? 8 ? 3 ? 5 ? 9 ?))).
```

「数独制約」述語をここでは，

- （第一条件）各空欄には 1 から n^2 までの数がはいること
- （第二条件）指定された n^2 個のマス（3 n^2 組ある）にはすべて異なる数がはいること

の 2 つによって表現する．

```
(数独制約 *array) ← (ListMembers *array),
                    (SD:diff *array).
```

ListMembers 述語は，第一の条件を記述している（付録 A 参照）．SD:diff 述語は，第 2 の条件を記述しており，「縦 (col)」,「横 (low)」,「箱 (box)」の 3 種類に分かれる．

```
(SD:diff *array) ← (SD:col *array),
                   (SD:low *array),
                   (SD:box *array).
```

各行の数が重複しないという条件を SD 述語で書くことにすると，SD:low 述語は そのまま SD 述語に帰着

できる（付録A参照）．SD:col 述語や SD:box 述語も，転置や並べ換えによって容易に SD 述語に帰着できる[2]．

SD 述語は,「リストの要素がすべて異なる」ことを意味する AllDifferent 述語に帰着できる．

```
(SD (*list . *rest)) ←
      (AllDifferent *list),
      (SD *rest).
(SD ()) ←.
```

AllDifferent 述語は,「対象がリストのどの要素とも等しくない」ことを意味する NotMember 述語に帰着され，さらに NotMember 述語は NotEqual 述語に帰着される（付録B参照）．

NotEqual 述語は,「２つの対象が等しくない」ことを意味する．この述語を宣言的プログラムで表現する１つの方法は，等しくない対象のすべての組合せに対する NotEqual 節（fact 節）を列挙することである．これは無限個あるが，宣言的プログラムの定義は無限個の節を許しているので問題はない．

```
(NotEqual 1 2) ←.
(NotEqual 3 5) ←.
        :
```

3 データ構造の拡大

3.1 一意的な展開

数独パズルを等価変換で解くには,「解答」節の body アトムを常に選んで，等価変換を繰り返せばよい．

最も基本的な等価変換の方法は，unfold 変換である．たとえば,「数独制約」述語のように，１つしか定義節を持たない述語の場合には，変換対象となるプログラムの節数を増加させないので，効率のよい等価変換がなされる．この等価変換を「一意展開」と呼ぶ．それを行う等価変換ルールは次のように書ける．

```
(Rule main
    (Head (解答 *array))
    (Body (初期配置 *array)
          (数独制約 *array))
```

このルールは (解答 *array) にマッチしたアトムを(初期配置 *array) と (数独制約 *array) の２つのアトムに書き換えることを意味する．

「数独制約」述語の他にも,「初期配置」,「SD:diff」,「SD:col」,「SD:low」,「SD:box」などの述語は定義節が１つなので同様に一意展開に使うことができる．

一方，ListMembers 述語 は２つの定義節を持つ．しかし数独パズルの場合には，ListMembers アトムの第一引数は長さ０以上のリストであり，唯一の節のヘッドにしか単一化しないので一意展開が可能である．このルールは 次のようになる．

```
(Rule LM1
    (Head (ListMembers ()))
    (Body))
(Rule LM2
    (Head (ListMembers (*list . *rest)))
    (Body (ListMember *list)
          (ListMembers *rest)))
```

これらのルールは，アトムが (ListMembers ()) ならばそれを取り除き，(ListMembers (*list . *rest)) にマッチするならば,

```
(ListMember *list)
(ListMembers *rest)
```

の列に書き直すことを意味する．これらのルールは，述語 ListMembers の引数が ニルまたはコンスのときにのみ発火できる．第一引数が 変数の場合には これらのルールは発火しない．それは ヘッドに対して (ユニフィケーションではなく) マッチングが使われるからである．これに対して アンフォールド変換の場合，単一化が用いられるので，アトムが 変数であっても展開が起こってしまう．

ListMember, SD, AllDifferent, NotMember などのアトムも，同様な意味で一意展開できる．

3.2 展開された結果のプログラム

上記の一意展開を可能なかぎり適用すると,「解答」節の body は，member アトム と NotEqual アトムだけになる．

ここで得られる member アトムの第二引数 は，常に (1 2 3 4 5 6 7 8 9) である．第一引数が ground であれば，その member アトムは真偽が定まり，そのアトムを変換対象プログラムから消去できる．そのルールは 次のように書ける．

```
(Rule mem1
    (Head (member *x *list))
    (Cond (ground *x)
          (ground *list))
    (Body (exec (member *x *list))))
```

このルールは，member の２つの引数が変数を含まないならば，実際に (member *x *list) の実行を行い，それが 成功すれば member アトムを取り除き，失敗すれば 節自体を取り除く[3]．同様なことは NotEqual アトムでもいえる．それらの引数が ground であれば，引数の

[2] 定義は 容易に想像でき，しかも煩雑で紙面を浪費するので 省略する．

[3] 解の存在する問題では，この段階では 失敗することはない．

比較によってそのアトムの真偽が判定でき，そのアトムを消去できる．

このような処理の結果，残されたアトムは，

```
(member *x (1 2 3 4 5 6 7 8 9))
(NotEqual *x 5)
(NotEqual 6 *y)
(NotEqual *x *y)
```

などの形のものばかりとなる．

3.3 member アトムの消去

この段階に来ると，ある1つのアトムに注目した一意的な変換は難しくなる．しかし2つ以上のアトムに注目すれば，一意的な変換が可能である．たとえば，次のような2つのアトムがボデイに含まれているとする．

```
(member *y (1 2 3 4 5 6 7 8 9))
(NotEqual 3 *y)
```

この場合，NotEqual アトムを見ると `*y` は 3 ではないから，member アトムを

```
(member *y (1 2 4 5 6 7 8 9))
```

に変更できるはずである．

しかしこのような変換はコストが高い．それは 変換対象となる2つのアトムを発見するためには，ボデイアトムの数 N の2乗のオーダーの計算量が必要となるからである．これを回避するためには，一方のアトムの情報を他方に効率的に伝達すればよい．そのために情報つき変数を用いる．

ここでは，member アトムの情報をボデイ全体に伝達することを考える．すなわち，

```
(member *y (1 2 3 4 5 6 7 8 9))
```

のようなアトムがあるとき，変数 `*y` に情報を持たせ，

```
*y~(1 2 3 4 5 6 7 8 9)
```

と変更する．ただし，この表現は，`1` から `9` までの整数からなるリストの要素である `*y` を意図したものである．このような変数はドメイン変数と呼ばれる [7]．

これを行うルールは次のように書ける．

```
(Rule mem2
    (Head (member *x *list))
    (Cond (pvar *x)
          (ground *x))
    (Body (putInfo *x *list)))
```

`(pvar *x)` は `*x` が純変数であることを判定する．`(putInfo *x *list)` は `*x` に `*list` をつけて情報つき変数にする．

純変数から情報つき変数への変化は瞬時に節全体に伝播する[4]．このルールが適用された結果 NotEqual アトムは，

```
(NotEqual 3 *y~(1 2 3 4 5 6 7 8 9))
```

にかわる．member アトムの持つすべての情報が情報つき変数によって表現されるので，member アトムは全部消去できる．以上の変換により，ボデイには，NotEqual アトムだけが残る．

4 NotEqual 述語の処理

4.1 述語 NotEqual に関する変換

残された NotEqual アトムを処理する方法を考えるのも，それほど難しくはない．たとえば，

```
(NotEqual *a~(1 2 4 5) *b~(4))
```

のように，情報が長さ1のリストである情報つき変数 `*b~(4)` を発見したとする．ドメイン変数の定義により，`*b` には1つの可能性 (`*b = 4`) しかないことがわかる．したがって 実際に`*b` を `*b = 4` に置き換えることができる．これを ETC で行うには，情報つき変数 `*b~(4)` から情報を取り除いて純変数にした後，その純変数 `*b` とリストの唯一の要素 `4` を単一化すればよい．その結果，NotEqual アトムは，

```
(NotEqual *a~(1 2 4 5) 4)
```

となる．

これを行うルールは次のように書ける．

```
(Rule mem3
    (Head (NotEqual *x *y~(*g)))
    (Body (exec (rmInfo *y)
                (= *y *g))
          (NotEqual *x *y)))
```

`*y` の情報を `rmInfo` でとり外して，`y` を `*g` にしている．情報つき変数の変化 `*y~(*g)` → `*g` は節全体に 自動的に伝播する．

また，NotEqual アトムの引数の片方が情報つき変数で，もう片方が数値である場合にも，容易に等価変換が可能である．たとえば，

```
(NotEqual *a~(1 2 4 5) 4)
```

の場合，`*a~(1 2 4 5)` を `*a~(1 2 5)` に変更して，NotEqual アトムを削除する．

これを行うルールは次のように書ける．

[4] それは 式中の 同じ変数の実体は 1つだからである．

```
(Rule mem4
    (Head (NotEqual *x *y))
    (Cond (number *y)
          (ivar *x)
          (getInfo *x *list))
    (Body (exec (remove *y *list *newlist)
                (putInfo *x *newlist))))
```

getInfo で *x の情報 *list を得て，remove で *list から y を取り除いたリスト *newlist を得ている．最後に *x の情報を *newlist に直している．

この他，引数が両方とも 数である場合の処理（3.2節参照）も用いる．以上の変換を順次施せば，問題1の答を得ることができる．

4.2 複数の可能性の場合分け

一般には，以上の変換だけでは，問題は解けない．それは，NotEqual アトムの両引数とも，長さ2以上のリストを情報に持つ情報つき変数である場合があるからである．もし，残った NotEqual アトムがすべてそのようなアトムである場合にはどうすればよいか．この場合，節を増加させずに変換を進めるのは難しいので，ここでは，最後の手段として 場合分けをすることにする．

たとえば，

```
(NotEqual *x~(2 3) *y~(2 5 7))
```

が残った場合，*x は 2 か 3 のいずれかなので，2 と仮定した場合と 3 と仮定した場合とに分けて変換を進める．これは，もとの節を*x = 2 あるいは *x = 3 で特殊化した2つの新しい節に置き換えて変換を進めることを意味する．

これを行うルールは次のように書ける．

```
(Rule mem5
    (Head (NotEqual *x~(*p *q) *y))
    (Body (exec (rmInfo *x)
                (= *x *p))
          (NotEqual *x *y))
    (Body (exec (rmInfo *x)
                (= *x *q))
          (NotEqual *x *y)))
```

rmInfo で *x の情報を取り除き，*x を *p にした場合と *q にした場合の2通りの節を作る．

5 処理時間の爆発

前章で示したルールを用いて実際に問題を解いた結果(処理時間とルール適用回数) を示す．

次に同じルールを用いて，問題のサイズを大きくした問題を扱う．次に扱う問題は 25×25 の数独パズルである．これを **問題2** (問題図は省略) と呼ぶ．この場合，盤面では 1 から 25 までの数字が入る．9×9 の問題の場合には 空白のマス目は 45 個程度のものが多いが，問題2の 空白のマス目は 325 個ある．

適用アトム	NotEqual
処理時間 (msec)	3 6 1 0
適用回数 (回)	1 8 6 4 0

表 1: 9×9 サイズの問題の処理の結果

一般的傾向として，空白の数が多くなるほど，問題を解くための計算量は多くなる．その様子を調べるために，問題2から 326 個の問題を 次のようにして作る．

まず 325 個の空白を ランダムに並べて，1 から 325 までの番号をふる．i 番目の空白を $blank_i$とする．次に問題2を解いて 各空白 $blank_i$ に入るべき数 num_i を求める．最後に，$m = 0, 1, 2, 3, \cdots, 325$ に対して，m 番目までを空白にし，$m+1$ 番目以降の すべての空白 $blank_i$ に数 num_i を埋めて新しい問題を作る．これにより 326 個の問題 $prob_m$ $(m = 0, 1, 2, 3, \ldots, 325)$ ができる．$Prob_0$ は 空白がない問題である．$Prob_i$ は 空白が i 個ある問題である．

図 2は，空白の数 i と 処理時間の関係を表すグラフである．グラフの横軸は，空白の数，縦軸は解が求まるまでの処理時間 (msec) を表している．

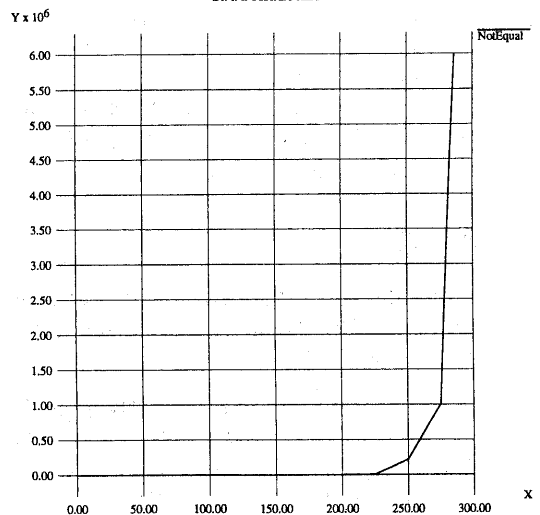

図 2: 空白の数と処理時間の関係

空白が 200 個程度までは 処理時間はほとんど変わらない．空白の数の数がそれ以上の場合，225 個で約 5 秒，259 個で約 3 分半，275 個で約 17 分，286 個で約 1 時間

40 分と，急激に時間がかかるようになる．これより，空白が 300 個以上の問題は 事実上解けないことがわかる．

これは場合分けがたくさん起こり，組合せの爆発が起こっているからである．これを解決するための方策については，紙面の都合上 別の機会に譲る．なお 概略は 文献 [10, 2] にある．

6 むすび

数独パズルを等価変換で解く方法を示した．この方法は 正当で高速な解法を構築するために適した構造をしている．それは，ルールを少しずつ追加し 新しいデータ構造を導入することでプログラムを改善することができ，個々の等価変換ルールの正当性を確保することによって全体のプログラムの正当性を厳密に保証することができるからである．

本論文の問題解決過程では，問題の仕様から，よいデータ構造，よいルール，よい制御が比較的自然に発見できている．

データ構造は，最初の仕様では，Prolog の項の範囲で書かれているが，異なる部分の情報を持ち寄るために，ドメイン変数というデータ構造が自然に導入された．これは，ETC に備えられた汎用のデータ構造である情報付変数によるものである．

上記のデータ構造の変換は，等価変換ルールによって行なわれる．また，NotEqual の制約変換も等価変換ルールによって行なわれる．それらのルールは，対象となる述語の定義から自然に思いつくものである．しかしこれらは，論理型のパラダイムでの推論の手続きによっては正当化できないルールである．

制御は，1つ以下の節しか生み出さないルールを優先し，2つ以上の節を生み出すルールの優先度を落とすことによって自然に実現されている．

参考文献

[1] 赤間清,繁田良則,宮本衛市:論理プログラムの等価変換による問題解決の枠組, 人工知能学会誌，Vol.12, No.2, pp.90–99 (1997).

[2] 赤間清，吹田慶子, 宮本衛市：データ構造の拡張とルールの追加による制約充足アルゴリズムの段階的改善, 日本ソフトウェア科学会第１４回大会論文集，辰口, E10-1, pp.529-532 (1997).

[3] 赤間清，繁田 良則，宮本衛市:特殊化システムの拡張による知識表現系の変更, 人工知能学会誌，Vol.13, No.1, pp.131–138 (1998).

[4] 赤間清，川口 雄一，宮本衛市：項領域における包含制約の等価変換, 人工知能学会誌，Vol.13, No.2, pp.112–120 (1998).

[5] 赤間清，川口 雄一，宮本衛市：論理的問題の等価変換による解法 (2), SLD 導出の限界, 人工知能学会誌，Vol.13, No.6, pp.936–943 (1998)

[6] 赤間清，清水伴訓，宮本衛市：宣言的プログラムの等価変換による問題解決, 人工知能学会誌，Vol.13, No.6, pp.944–952 (1998)

[7] Hentenryck, V. : Constraint Satisfaction in Logic Programming, The MIT Press (1989).

[8] Lloyd, J.W.: *Foundations of Logic Programming*, Second edition, Springer-Verlag (1987)

[9] 清水 伴訓, 赤間清，宮本衛市：等価変換プログラミング言語 ETC, 電子情報通信学会 ソフトウエアサイエンス研究会, SS 96-19, pp.9-16 (1996)

[10] 吹田慶子, 赤間清，宮本衛市：等価変換による制約充足問題の解法, 電子情報通信学会 ソフトウエアサイエンス研究会, SS 96-18, pp.1-8 (1996)

付録 A

数独パズルの宣言的プログラムによる定式化に必要な節のうち，いくつかを補う．

```
(ListMembers ()) ←.
(ListMembers (*list . *rest)) ←
      (ListMember *list),
      (ListMembers *rest).

(ListMember ()) ←.
(ListMember (*x . *rest)) ←
      (member *x (1 2 3 4 5 6 7 8 9)),
      (ListMember *rest).

(member *x (*x . *R)) ←.
(member *x (*y . *R)) ← (member *x *R)

(SD:low *array) ← (SD *array).

(AllDifferent ()) ←.
(AllDifferent (*x . *rest)) ←
      (NotMember *x *rest),
      (AllDifferent *rest).

(NotMember *x ()) ←.
(NotMember *x (*y . *rest)) ←
      (NotEqual *x *y),
      (NotMember *x *rest).
```