TM-0468

有限領域を対象とした単一化の拡張
(Finite Domainの試作)

近藤省造，今村　誠

March, 1988

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

1．はじめに

近年の論理型言語に関する研究には，Prologの拡張による新しい論理型言語の提案が多い。その中心的なものとして，次の２つが挙げられる。

1) 並列型論理言語の提案
2) 制約型論理言語の提案

1)と2)は，データ駆動型の計算メカニズム(data-driven computation)という点で密接なかかわりを持つが，ここでは制約プログラミング(constraint programming)の観点から後者に注目する。

制約型論理言語の研究には，論理型言語と関数型言語の融合やデータタイプの導入など幾つかのアプローチがある。なかでも，Prologの基本機能である単一化(unification)をEquality Constraint Solver(以下ＥＣＳと略す)とみなし，これを拡張する形でより強力なConstraint Solver(例えば１＋２と３を単一化したり，１＋Ｘと３を単一化してＸ＝２を得る)を実現しようという研究が最近盛んに行われている。

これらの研究の多くが，遅延実行メカニズムを用いて，評価可能となった時点で制約チェックが起動するという，データ駆動型の計算メカニズムを提供している。例えば，制約：１＋Ｘ＝Ｙに対し

a) ＸとＹが共に具体化(instantiate)されていない状態では制約チェックが遅延され，ＸとＹの少なくとも一方が具体化された時点で制約チェックが起動する。
b) ＸとＹの少なくとも一方が既に具体化されている時点では，制約チェックは遅延されることなく即実行される。

これに対し，Dincbasの有限領域を対象とした単一化の拡張[Dincbas 87][奥西 87][川村 87]では，単一化を変数の値と領域を対象としたＥＣＳに拡張することで，従来は評価可能になって初めて評価されていた制約をより積極的に利用しようという立場をとっている。特に，この拡張は制約充足問題(Constraint Satisfication Problem)に有効とされている。

2．Dincbas の拡張の概要

従来の制約型論理言語において制約はいわば受動的に扱われていた。即ち，制約は評価可能になって初めて評価され，矛盾（失敗）が起こればバックトラックして別解を探すというものであった。この様な後向き推論(backward checking)はPrologにおける基本メカニズムである。これに対しDincbasの有限領域を対象とした単一化の拡張では，制約を能動的に扱う前向き推論(forward checking)によって，不要なバックトラックの回避を目指している。

Prologにおける変数の領域が(陰に)エルブラン領域に規定されているのに対し，変数の領域を(陽に)エルブラン領域のサブセットとして規定するのが有限領域(finite domain)である。制約充足問題においては変数の領域は有限でかつ予め与えられているため，解となる組合せの検出アルゴリズムを，エルブラン領域よりも，与えられた領域に適用した方が有効である。

例えば，

変数Ｘには［１，５］　（１以上５以下の自然数）
変数Ｙには［１，３］　（１以上３以下の自然数）

と領域が与えられている場合，

制約：Ｘ＋Ｙ＝８を満足するＸとＹの値は一意（Ｘ＝５，Ｙ＝３）に決定し，

制約：Ｘ＋Ｙ＝７を満足するＸとＹの領域はＸ：［４，５］，Ｙ：［２，３］に絞り込まれる。これは，制約プログラミングの観点から見れば，各変数に領域という一種の制約が与えられている場合と解釈できる。

ここにおいて単一化は，変数に対する代入としてその値を求めるだけでなく，変数の領域も求めるように拡張される。具体的には，以降で述べる

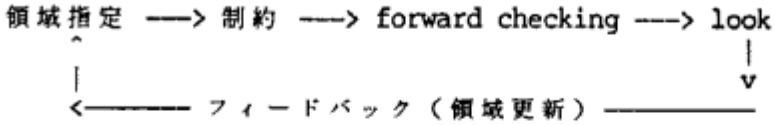
1) 領域付き変数の導入 と
2) その変数に対する単一化 及び unequality などの組込み述語

によって実現される。

試作にあたってのポイントは

a) 領域付き変数の実現法
b) unificationの拡張 と unequalityの定義
c) 前向き推論(forward checking) と look ahead

で，特に以下に示すようなフィードバック・メカニズムは重要である。



つまり，プログラム実行中に変数に値が代入された時点だけでなく，領域が更新された時点でも制約のチェック（及びそれに続くforward checking, look ahead）を連動させ，プログラムの実行中は常に制約がきいてくるようなメカニズムが必要である。言い換えれば，プログラム中の全ての制約を満足させる様な解釈(interpretation)を導き出すメカニズムである。従って，本試作は制約プログラミングのための一種の言語処理系を設計することに相当する。

尚，本試作には逐次型推論マシンPSI 上の CIL V2.0 [ＣＩＬ 87]を使用した。

3．領域付き変数の実現法

3.1 領域付き変数

以下のシンタックスを持つ構造体として領域付き変数(d-variable)を定義する。
h-variableはPrologの通常の変数である。

```
      V :: Domain
      |       |
      |     領域 ―― D_list(Difference List) で実現
    h-variable
```

領域が更新される毎に，D_listも更新される。

```
      V::[[0,1,2,3,4,5,6,7,8,9],[2,3,4], ... ]
      C::[[green,yellow,red,blue],[red,blue], ... ]
```

h-variableと領域の間には同期がとれており，実行中に領域が1つの値に絞られるとh-variableはその値に具体化され，逆にh-variableが具体化されると領域はその値のみとなる。

```
      3::[[0,1,2,3,4,5,6,7,8,9],[2,3,4],[3,4],[3]]
    red::[[green,yellow,red,blue],[red,blue],[red]]
```

3.2 領域更新に伴う制約チェックの再起動

領域が更新された場合に制約チェックを再起動させるため，
1) d-variableが具体値をまだ持たない場合，D_listの最後尾を変数としておき，

```
      V::[[0,1,2,3,4,5,6,7,8,9],[2,3,4]|Next]
```

2) 領域が更新された（D_listの最後尾が具体化された）時点で，次ステップが起動するようにCILの組込み述語freezeを使用している。

```
      D = [ ... |Next],
      freeze(Next, Constant \= V::D)
```

具体例は unification 及び unequality のプログラムで示す。

4．unificationの拡張とunequalityの定義

領域付き変数(d-variable)の導入に伴って，unificationの拡張とunequalityの定義を行う。unificationについては，Prolog(厳密にはCIL)のそれを，単一化処理によって変数領域の継承や領域間での相互作用が生じるように拡張する。具体的な定義を以下に示す。

1) h-variableとd-variableを単一化すると，h-variableはそのd-variableの領域と同じ領域を持つd-variableとなる。（領域の継承）
2) 定数とd-variableを単一化すると
   a) 定数がd-variableの領域に含まれていれば，d-variableの変数部（D_listの最後尾）に定数が代入され，成功する。
   b) 定数がd-variableの領域に含まれていなければ失敗する。
3) d-variable(X::Dx,Y::Dy)同士を単一化すると，それらの共通領域をDとし，
   a) Dが空ならば失敗する。
   b) D = {v}（要素数が1）ならばX = Y = vで，成功する。
   c) その他の場合は，X = Yとし，DxとDyをDに更新して，成功する。

unificationの定義

以下は，これを実現する述語(d_variable_unify)の定義例である。
尚，6章で示す例題プログラムではオペレータ =で呼び出される。

```
% ---------- d_variable_unify ----------

d_variable_unify(X, Y) :- (h_variable_p(X) ; h_variable_p(Y)), !, unify(X, Y).
d_variable_unify(X::Dx, Y::Dy) :- (var(Dx) ; var(Dy)), !, Dx = Dy, X = Y.
d_variable_unify(X::Dx, Y::Dy) :- var(X), var(Y), nonvar(Dx), nonvar(Dy), !,
        get_domain(Dx, Rx, _), get_domain(Dy, Ry, _),
        intersection(Rx, Ry, Range),
       (Range == [], !, fail
          ;    replace_domain(X, Dx, Range, _),
               replace_domain(Y, Dy, Range, _),
              (Range = [_], !, true
                 ;  get_domain(Dx, _, X_next),
                    get_domain(Dy, _, Y_next),
                    freeze(X_next, Y_next, d_variable_unify(X::Dx, Y::Dy)))).
d_variable_unify(X::Dx, Y::Dy)  :- nonvar(X), !,
       (nonvar(Dy), !,
            get_domain(Dy, Ry, _), member(X, Ry),
            replace_domain(Y, Dy, [X], _)
         ;  Dy = Dx, Y = X).
d_variable_unify(Y::Dy, X::Dx)  :- nonvar(X), !,
       (nonvar(Dy), !,
            get_domain(Dy, Ry, _), member(X, Ry),
            replace_domain(Y, Dy, [X], _)
         ;  Dy = Dx, Y = X).
d_variable_unify(X::_, Y::_)    :- nonvar(X), nonvar(Y), X == Y.
d_variable_unify(X, Y::Dy)      :- constant(X), !,
       (nonvar(Dy), !,
            get_domain(Dy, Ry, _), member(X, Ry),
            replace_domain(Y, Dy, [X], _)
         ;  d_variable(Y, [X], Y::Dy)).
d_variable_unify(Y::Dy, X)      :- constant(X), !,
```

```
        (nonvar(Dy), !,
             get_domain(Dy, Ry, _), member(X, Ry),
             replace_domain(Y, Dy, [X], _)
          ;  d_variable(Y, [X], Y::Dy)).
d_variable_unify(X, Y)          :- constant(X), constant(Y), X == Y.
```

一方，unequality(\=)は次の様に定義される。

1) Xがd-variable(X::Dx)で，Yが定数ならばDxからYを取り除いた領域をDとし，
   a) D = {v} ならば X = v として，成功する。
   b) それ以外ならば，Dx を D に更新して，成功する。
2) Xが定数で Yがd-variable(Y::Dy)の場合は，それぞれを置き換えて(1)と同様
3) X と Yが定数の場合は，そのまま X \= Y を実行する。

unequality の定義

以下は，これを実現する述語(d_variable_unequal)の定義例である。
尚，6章で示す例題プログラムではオペレータ \= で呼び出される。

```
% ---------- d_variable_unequal ----------

d_variable_unequal(X::Dx, Y::Dy) :-
        var(X), nonvar(Dx), var(Y), nonvar(Dy),
        get_domain(Dx, Rx, _), get_domain(Dy, Ry, _),
        intersection(Rx, Ry, D), D == [], !.
d_variable_unequal(X::Dx, Y::Dy) :-
        var(X), nonvar(Dx), var(Y), nonvar(Dy), !,
        get_domain(Dx, _, X_next),
        get_domain(Dy, _, Y_next),
        freeze(X_next, Y_next, d_variable_unequal(X::Dx, Y::Dy)).
d_variable_unequal(X::Dx, Y)     :-
        var(Dx), !, freeze(Dx, d_variable_unequal(X::Dx, Y)).
d_variable_unequal(Y, X::Dx)     :-
        var(Dx), !, freeze(Dx, d_variable_unequal(Y, X::Dx)).
d_variable_unequal(X::_, Y::Dy)  :-
        nonvar(X), var(Y), nonvar(Dy),
        get_domain(Dy, Ry, _), not_member(X, Ry), !.
d_variable_unequal(X::_, Y::Dy)  :-
        nonvar(X), var(Y), nonvar(Dy), !,
        get_domain(Dy, Ry, _), remove(X, Ry, Range),
       (Range == [], !, fail ;
        replace_domain(Y, Dy, Range, _)).
d_variable_unequal(Y::Dy, X::_)  :-
        nonvar(X), var(Y), nonvar(Dy),
        get_domain(Dy, Ry, _), not_member(X, Ry), !.
d_variable_unequal(Y::Dy, X::_)  :-
        nonvar(X), var(Y), nonvar(Dy), !,
        get_domain(Dy, Ry, _), remove(X, Ry, Range),
       (Range == [], !, fail ;
        replace_domain(Y, Dy, Range, _)).
d_variable_unequal(X::_, Y::_)   :- constant(X), constant(Y), X \== Y.
d_variable_unequal(X, Y::Dy)     :-
        constant(X), nonvar(Dy),
        get_domain(Dy, Ry, _), not_member(X, Ry), !.
```

```
d_variable_unequal(X, Y::Dy)      :-
        constant(X), nonvar(Dy),
        !, get_domain(Dy, Ry, _), remove(X, Ry, Range),
       (Range == [], !, fail ;
        replace_domain(Y, Dy, Range, _)).
d_variable_unequal(Y::Dy, X)      :-
        constant(X), nonvar(Dy),
        get_domain(Dy, Ry, _), not_member(X, Ry), !.
d_variable_unequal(Y::Dy, X)      :-
        constant(X), nonvar(Dy), !,
        get_domain(Dy, Ry, _), remove(X, Ry, Range),
       (Range == [], !, fail ;
        replace_domain(Y, Dy, Range, _)).
d_variable_unequal(X, Y)          :- constant(X), constant(Y), X \== Y.
```

5．前向き推論 と look ahead

5.1 前向き推論

= や \= の関係付けられた変数同士は，まだ値が具体化していなくとも，それぞれの領域情報を用いることによって，互いの領域を狭めることができる。
——> unification の定義の 1),3) や unequality の定義の 1)の b)
更に，場合によっては，値が一意に決まることもある。
——> unification の定義の 2)の a), 3) の b) や unequality の定義の 1) の a)
これらを総称して，前向き推論(forward checking)と呼ぶ。

5.2 look ahead

制約が四則演算から成る計算式などで与えられる場合，= や \= に対する前向き推論で得られた「演算結果（ダミー変数）に対する新しい領域」を，計算式にフィードバックさせて式中の変数の領域を狭めることが可能な場合がある。このフィードバック・メカニズムを look aheadと呼ぶ。
更に，look ahead の結果が再び前向き推論を呼び出すという具合に両者は循環し，式中の全変数が具体化されるまでこれが繰り返される。 また，前向き推論や look ahead の起動は共通の変数を使っている他計算式における処理の副作用として発生する場合もある。

（加算における look ahead の例）

```
計算式：      A+B+C= ...      (A,B,C: d-variable)
領域：        A:[MinA1,MaxA1], B:[MinB1,MaxB1], C:[MinC1,MaxC1]
左辺の和：    X:[MinX1,MaxX1] ===> X:[MinX2,MaxX2]
                                  （前向き推論の結果）

各変数の新しい領域：
A:[MinA2,MaxA2] = [MinX2-max(B+C),MaxX2-min(B+C)] ∧ [MinA1,MaxA1]
B:[MinB2,MaxB2] = [MinX2-max(A+C),MaxX2-min(A+C)] ∧ [MinB1,MaxB1]
C:[MinC2,MaxC2] = [MinX2-max(A+B),MaxX2-min(A+B)] ∧ [MinC1,MaxC1]
                        （look ahead の結果）
```

以下に具体例とそのプログラム例を示す。

```
計算式：      A+B = C+D
領域：        A:[1,5], B:[2,3], C:[2,4], D:[3,7]
左辺の和：    X:[3,8]  ===> X:[5,8]
右辺の和：    Y:[5,11] ===> Y:[5,8]
                          （前向き推論の結果）

各変数の新しい領域：
A:[2,5] = [5-3,8-2] ∧ [1,5] = [2,6]  ∧ [1,5]
B:[2,3] = [5-5,8-1] ∧ [2,3] = [0,7]  ∧ [2,3]
C:[2,4] = [5-7,8-3] ∧ [2,4] = [-2,5] ∧ [2,4]
D:[3,6] = [5-4,8-2] ∧ [3,7] = [1,6]  ∧ [3,7]
```

プログラム例：
述語 d_variable は第１引数の変数に第２引数の領域を割り付け，第３引数に領域付き変数を返すものである。

```
sample(A1, B1, C1, D1) :-
    d_variable(A, [1,2,3,4,5],  A1),
    d_variable(B, [2,3],        B1),
    d_variable(C, [2,3,4],      C1),
    d_variable(D, [3,4,5,6,7],  D1),
    A1 + B1 = C1 + D1.
```

実行例（on CIL V2,0）：

```
sample(A,B,C,D).

A => E::[[1,2,3,4,5],[2,3,4,5]|F]
B => E::[[2,3]|F]
C => E::[[2,3,4]|F]
D => E::[[3,4,5,6,7],[3,4,5,6]|F]
yes
```

6．例題

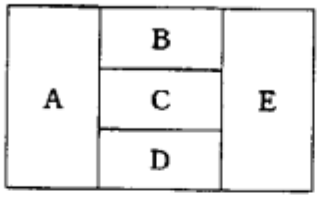
本章では，Dicbas の単一化の拡張を実際の問題に適用した例を示す。
問題は，パズルとしてよく知られている

1) 色塗り問題
2) 覆面算（SEND + MORE = MONEY）
3) 8クィーンパズル

である。

6.1 色塗り問題

色塗り問題とは，数枚のタイルが互いに隣接したものと異なる色で塗り分けられるように有限個の色を割り振るもので，ここで扱うのは下図に示すような5枚のタイルに緑(green)，黄(yellow)，赤(red)，青(blue)の4色を塗り分ける問題である。



色塗り問題

以下に，この問題に対応したプログラムとその実行例を示す。print_domain は各変数の領域を画面上に表示する述語で，labelingは第1引数（リスト）の各要素（変数）に対し，与えられている領域から適当な値を選びだして割り付ける述語である。尚，このプログラムでは，labelingの内部で変数への割り付けが実施される毎に，第2引数（リスト）の各要素（変数）に対する，その時点での領域を表示している。

```
% ------- problem: color -------

color([A,B,C,D,E]) :-
        d_variable(A, [green,yellow,red,blue], A1),
        d_variable(B, [green,yellow,red,blue], B1),
        d_variable(C, [green,yellow,red,blue], C1),
        d_variable(D, [green,yellow,red,blue], D1),
```

```
        d_variable(E, [green,yellow,red,blue], E1),
        nl,
        print_domain([A1,B1,C1,D1,E1]),
        A1 \= B1,
        A1 \= C1,
        A1 \= D1,
        B1 \= C1,
        B1 \= E1,
        C1 \= D1,
        C1 \= E1,
        D1 \= E1,
        labeling([A1,B1,C1,D1,E1], [A1,B1,C1,D1,E1]).
```

実行例では，最初に各変数に対する領域の初期状態，次にＡに緑が割り付けられた時の各変数の領域状態，以下，Ｂに黄，Ｃに赤，Ｄに黄，Ｅに緑の順に割り付けが行われた時の各変数の領域状態が表示されている。

```
color(A).
                        % 変数領域の初期状態
[green,yellow,red,blue]
[green,yellow,red,blue]
[green,yellow,red,blue]
[green,yellow,red,blue]
[green,yellow,red,blue]
                        % Ａに緑を割り当てた時点での変数領域の状態
[green]
[yellow,red,blue]
[yellow,red,blue]
[yellow,red,blue]
[green,yellow,red,blue]
                        % Ｂに黄を割り当てた時点での変数領域の状態
[green]
[yellow]
[red,blue]
[yellow,red,blue]
[green,red,blue]
                        % Ｃに赤を割り当てた時点での変数領域の状態
[green]
[yellow]
[red]
[yellow,blue]
[green,blue]
                        % Ｄに黄を割り当てた時点での変数領域の状態
[green]
[yellow]
[red]
[yellow]
[green,blue]
                        % Ｅに緑を割り当てた時点での変数領域の状態
[green]
[yellow]
[red]
[yellow]
[green]
```

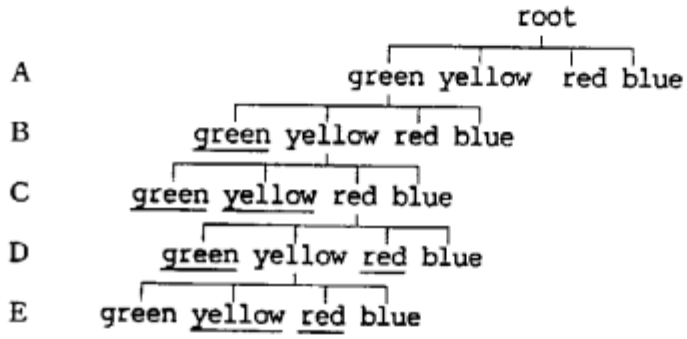
```
A => [green,yellow,red,yellow,green] ;          % 1個目の解

[green]
[yellow]
[red]
[yellow]
[blue]                    % Eには次候補として青しか残されていない

A => [green,yellow,red,yellow,blue]             % 2個目の解

yes
```

この問題を制約付き探索問題とみなして，プログラム実行過程を木構造で表現したものが下図であり，実行中の領域の減少は探索木の枝刈りに対応している。下線部が刈られた枝の部分を示している。従って，Aに緑，Bに黄，Cに赤，Dに黄を割り付けた時に，Eの候補として残されている色は緑と青の2色だけである。



6.2 覆面算（SEND + MORE = MONEY)

ここで扱う覆面算は，SEND + MORE = MONEY と呼ばれるもので，

「以下の加算を成立させるように各英文字に0から9を割り当てる。

```
   S E N D
+  M O R E
-----------
 M O N E Y
```

但し，各英文字には異なる数を割り当てるものとする。」

である。制約としては，「各英文字の取りうる数の範囲」，「計算式」，「他と異なる数を取る」の3種類が与えられている。

以下に，この問題に対応したプログラムとその実行例を示す。述語 alldifferentは各英文字（変数）を「他と異なる」という制約で関連付けるもので，述語 statisticは実行時間を計測する為のＣＩＬ組込み述語である。

```
% ------- problem: send + more = money -------

sendmory(X, T):-
    X = [[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]],
    statistics(runtime,_),
    d_variable(S, [0,1,2,3,4,5,6,7,8,9], Sd),
    d_variable(E, [0,1,2,3,4,5,6,7,8,9], Ed),
    d_variable(N, [0,1,2,3,4,5,6,7,8,9], Nd),
    d_variable(D, [0,1,2,3,4,5,6,7,8,9], Dd),
```

```
    d_variable(M, [0,1,2,3,4,5,6,7,8,9], Md),
    d_variable(O, [0,1,2,3,4,5,6,7,8,9], Od),
    d_variable(R, [0,1,2,3,4,5,6,7,8,9], Rd),
    d_variable(Y, [0,1,2,3,4,5,6,7,8,9], Yd),
    d_variable(R1, [0,1], R1d),
    d_variable(R2, [0,1], R2d),
    d_variable(R3, [0,1], R3d),
    d_variable(R4, [0,1], R4d),
    alldifferent([Sd,Ed,Nd,Dd,Md,Od,Rd,Yd]),
                                          %  alldifferent([S,E,N,D,M,O,R,Y])
    Sd \= 0,                               %  S \= 0
    Md \= 0,                               %  M \= 0
    R1d = Md,                              %  R1 = M
    R2d + Sd + Md = Od + 10 * R1d,         %  R2 + S + M = O + 10 * R1
    R3d + Ed + Od = Nd + 10 * R2d,         %  R3 + E + O = N + 10 * R2
    R4d + Nd + Rd = Ed + 10 * R3d,         %  R4 + N + R = E + 10 * R3
    Dd + Ed = Yd + 10 * R4d,               % D + E = Y + 10 * R4
    nl,
    print_domain([Sd,Ed,Nd,Dd,Md,Od,Rd,Yd,R1d,R2d,R3d,R4d]),
    labeling([Sd,Ed,Nd,Dd,Md,Od,Rd,Yd,R1d,R2d,R3d,R4d],
             [Sd,Ed,Nd,Dd,Md,Od,Rd,Yd,R1d,R2d,R3d,R4d]),
    statistics(runtime,T).

alldifferent([]):- !.
alldifferent([X|Xs]):- outof(Xs, X), alldifferent(Xs).

outof([], _):- !.
outof([Y|Ys], X):- X \= Y, outof(Ys, X).
```

実行例では，まず制約によって各英文字の領域が絞り込まれ，次にその領域から述語 labeling が解を探索する。プログラムの実行結果は，述語 sendmoryの第１引数に具体化されたリストで，リストの要素は各英文字に割り当てられた数を示している。つまり，S=9,E=5,N=6,D=7,M=1,O=0,R=8,Y=2 が解である。 次候補（別解）は存在しない。尚，第２引数には実行時間（単位はmsec）が具体化されている。

```
sendmory(X,T).
                        % 制約によって絞り込まれた各英文字の領域
[9]                     % S の領域
[2,3,4,5,6,7,8]         % E の領域
[2,3,4,5,6,7,8]         % N の領域
[2,3,4,5,6,7,8]         % D の領域
[1]                     % M の領域
[0]                     % O の領域
[2,3,4,5,6,7,8]         % R の領域
[2,3,4,5,6,7,8]         % Y の領域
[1]                     % R 1 の領域
[0]                     % R 2 の領域
[0,1]                   % R 3 の領域
[0,1]                   % R 4 の領域

T => [A,5067]
X => [[9,5,6,7],[1,0,8,5],[1,0,6,5,2]]
yes
```

6.3 8クィーンパズル

8クィーンパズルは「チェス盤の上に8個のクィーンをお互いに取られないように並べる」という問題であり，クィーンが将棋の角と飛車を足した動きをすることから「互いの駒の縦，横，斜めのライン上に位置しないように8個のクィーンを並べる」と置き換えられる。更に，駒と駒の位置関係に注目してみると，互いに相手のライン（直線）上に位置しないという単純な制約が幾重にも絡み合って複雑な組合せ問題を形成している事に気付く。しかし直線上に位置しないという制約は，盤上のある位置（X，Y）にクィーンを置いた時，簡単な式を使って

制約1：座標Xに他の駒が位置していない
制約2：座標Yに他の駒が位置していない
制約3：座標Xと座標Yの和が一致する座標に他の駒が位置していない
制約4：座標Xと座標Yの差が一致する座標に他の駒が位置していない

と表現できる。つまり8クィーンパズルは「この4種類の制約を常に満足する各クィーンの座標を求めること」と置き換えられる。但し，8クィーンパズルの舞台であるチェス盤上という限られた世界では，各駒の取りうる座標にはあらかじめ「1から8まで」という領域が与えられている。

以下に，この問題に対応したプログラムとその実行例を示す。各クィーンのX座標にはあらかじめ1から8が割り当てられている。述語make_queen_rangeは各クィーンのY座標の初期値域（1から8までの整数を要素とするリスト構造で表現）を生成する。　述語setup_queenでは，各クィーンのX座標，Y座標，その和，その差に対応した値域付き変数を生成する。述語alldifferentで各クィーンのX座標，Y座標，X座標とY座標の和，差が他クィーンのそれぞれと異なるように関係付けている。述語labelingでは各クィーンのY座標にその値域から順に選び出した値を割り当てる。この割り当てに連動して，制約計算が起動し，制約を満たす割り当てか否かの判定と動的な値域の絞り込みが実施される。

```
eight_queen(Ans) :-
    Ans = [[1,A], [2,B], [3,C], [4,D], [5,E], [6,F], [7,G], [8,H]],
    make_queen_range(1, 8, Range),
    setup_queen(Ans, List_Y, List_add_XY, List_sub_XY, Range),
    alldifferent(List_Y),
    alldifferent(List_add_XY),
    alldifferent(List_sub_XY),
    labeling(List_Y, List_Y).

make_queen_range(N, N, [N]) :- !.
make_queen_range(N, M, [N|L]) :- make_queen_range(N + 1, M, L).

setup_queen([], [], [], [], _) :- !.
setup_queen([[X, Y]|L], [Yd|L1], [A|L2], [S|L3], Range) :-
    d_variable(X, [X], Xd),
    d_variable(Y, Range, Yd),
    Xd + Yd = A,
    Xd - Yd = S,
    setup_queen(L, L1, L2, L3, Range).
```

実行例では，途中経過として，各クィーンのY座標の領域が実行中に変化している様子も表示している。ちなみに，制約を満足する解の総数は 92個であった。

```
eight_queen(L).
```

```
=>[1]                    % (1,1) に駒を置く
  [3,4,5,6,7,8]
  [2,4,5,6,7,8]
  [2,3,5,6,7,8]
  [2,3,4,6,7,8]
  [2,3,4,5,7,8]
  [2,3,4,5,6,8]
  [2,3,4,5,6,7]

    [1]
  =>[3]                  % (2,3) に駒を置く
    [5,6,7,8]
    [2,6,7,8]
    [2,4,7,8]
    [2,4,5,8]
    [2,4,5,6]
    [2,4,5,6,7]

        % (3,5) に駒を置くと，残りの駒をどこに置いても，将来
        % 制約を満足しなくなる事が，この時点の計算で明らかなため
        % (3,5)はスキップされる。

      [1]
      [3]
    =>[6]                % (3,6) に駒を置く
      [2,8]
      [2,7]
      [2,4,5,8]
      [4,5]
      [2,4,5,7]

        % (4,2),(4,8)のどちらに駒を置いても，残りの駒を置く時点で
        % 制約が満足されない事が，この時点の計算で明らかなため
        % (3,6)は放棄される。

      [1]
      [3]
    =>[7]                % (3,7) に駒を置く
      [2]
      [4,8]
      [5,8]
      [4,6]
      [4,5]

        [1]
        [3]
        [7]
      =>[2]              % (4,2) に駒を置く
        [4,8]
        [5,8]
        [4,6]
        [4,5]
```

```
      % (5,4),(5,8)のどちらに駒を置いても，残りの駒を置く時点で
      % 制約が満足されない事が，この時点の計算で明らかなため
      % (4,2)は放棄される。更に，(4,2)には次候補がないため(3,7)も
      % 放棄される。

    [1]
    [3]
  =>[8]                 % (3,8) に駒を置く
    [2,6]
    [2,4,7]
    [2,4]
    [2,5,6]
    [2,4,5,6,7]

      % (4,2),(4,6)のどちらに駒を置いても，残りの駒を置く時点で
      % 制約が満足されない事が，この時点の計算で明らかなため
      % (3,8)は放棄される。更に，(3,8)には次候補がないため(2,3)も
      % 放棄される。

  [1]
=>[4]                   % (2,4) に駒を置く
  [2,6,7,8]
  [3,5,7,8]
  [2,3,6,8]
  [2,3,5,7]
  [2,3,5,6,8]
  [2,3,5,6,7]

    [1]
    [4]
  =>[2]                 % (3,2) に駒を置く
    [5,7,8]
    [3,6,8]
    [3,7]
    [3,5,8]
    [3,5,6]

      . . . 以下 同様にして処理が進み . . .

    [1]
    [5]
  =>[8]                 % (3,8) に駒を置く
    [2,6]
    [3,4,7]
    [2,3,4,7]
    [2,3,6]
    [2,4,6,7]

      [1]
      [5]
      [8]
    =>[2]               % (4,2) に駒を置く
      [4,7]
      [3,7]
```

```
   [3,6]
   [4,7]

   [1]
   [5]
   [8]
  =>[6]              % (4,2)は放棄され，(4,6) に駒を置く
   [3]
   [7]
   [2]
   [4]

   % この時点で残りの駒のＹ座標は既に決定している。

    [1]
    [5]
    [8]
    [6]
   =>[3]             % (5,3) に駒を置く
    [7]
    [2]
    [4]

      [1]
      [5]
      [8]
      [6]
      [3]
     =>[7]          % (6,7) に駒を置く
      [2]
      [4]

        [1]
        [5]
        [8]
        [6]
        [3]
        [7]
       =>[2]        % (7,2) に駒を置く
        [4]

          [1]
          [5]
          [8]
          [6]
          [3]
          [7]
          [2]
         =>[4]      % (8,4) に駒を置く

L => [[1,1],[2,5],[3,8],[4,6],[5,3],[6,7],[7,2],[8,4]]
                    % 1 個目の解
yes
```

7．おわりに

本稿で紹介した「有限領域を対象とした単一化の拡張」は制約型論理言語の研究として位置付けられるものであり，その応用としては探索問題における探索アルゴリズムの効率化（静的／動的な枝刈りを行ない，解の候補を絞り込むと共に不要なバックトラックを防ぐ）などが考えられる。より一般的には，変数に対する領域を与えた際のプログラムの最適化技術としてコンパイラにも応用可能である。インプリメンテーション上の問題点としては，領域付き変数の実現方式と領域計算（intersection 等）の効率化が挙げられる。

－ 参考文献 －

[Dincbas 87] M.Dincbas etc, Extending Equation Soliving and Constraint Handling in Logic Programming, ECRC Internal Report IR-LP-2202, Feb 1987

[奥西 87] 奥西，ロジックプログラミングにおける等式ソルビングと制約処理の拡張（上記抄訳）,CIL-Meeting Memo, ICOT 2Lab.

[川村 87] 川村 他，CS-Prolog:拡張単一化基づくCONSTRAINT SOLVER, Proceedings of Logic Programming Conference '87, pp21-28

[ＣＩＬ 87] ＣＩＬ言語マニュアル 第２版 第１刷，Mar 1987, ICOT