論理プログラムの変換による効率化

中川　裕志
NAKAGAWA HIROSHI
（横浜国立大学　工学部　情報工学科）

はじめに

プログラム変換の研究に関しては、最近にACM のComputing Surveys に"Program Transformation Systems"という立派なサーベイ［１］が書かれ、またＢＩＴ誌に黒川、外山氏によって　"プログラム変換を論じる"　という明快な記事が発表されている。プログラム変換は大別して（1)述語論理などで表現された仕様から計算可能な論理プログラム（必ずしも効率は良くない。）を導出するもの（Hogger[2], Bible[3], Sato[4]）、（2)効率の良くない宣言的な論理プログラムから変換により手続き的な効率良いプログラムを得るもの（Tamaki[5], Tarnland[6], Kahn[7], Takesima[8], Nakamura[9], Reddy[10]）、（3)関数型プログラムに対して効率化を行なうもの（Burstall[11], Darlington[12],[13], Wand[14] ）に分類されるであろう。（多分に筆者の主観が混入しているとは思うが）このように多数の研究があるにもかかわらず、データ構造を扱うアルゴリズムに関しては、前記の [6],[13] に（ｄ－）リストに関連するものがあるが、あまり大きな成果は得られていないようである。本論文では、(2）の路線に乗ってデータ構造、特に木構造を扱ってみようとする試みに関して述べる。

１．木構造の認識

大規模なデータを扱うために、まずこれを認識する必要があるわけだが、人間がどのようにしてこの認識と処理を行なっているかは、我々にとって参考になろう。周知のように、言語であれ画像であれ人間の認識プロセスは短期記憶を用いたプロダクションシステム的なものと思われている。ただし短期記憶の容量は、magical numberとよばれる７±２で抑えられている。そこで、それ以上大きな情報はポインタ化していると言われている。すなわち適当な大きさにグループ化して階層的に記憶し処理をしている。人間が短期記憶上の情報を処理する時には、必要なchunk のみをアクセスしさらに展開したりして情報処理を行ない、無関係なあるいはその処理によって不変なchunk は手をつけないまま保持される。階層的な記憶方法は計算機において重用されている木構造とほとんど相似なものである。したがって木構造を扱うアルゴリズムの自動合成を考えるにあたっては上記のことは大きなヒントをあたえてくれる。たとえばある木にNIL を挿入する、すなわちなにも挿入しないなら、その木の構造は不変であるーーＨ１，同じ結果が得られる計算すなわち再計算は避けて、元のchunk つまり部分木の構造を保存するーーＨ２，といったheuristic が浮かんでくる。これらのheuristic をPrologプログラムの変換のドメインで考えてみるなら、現実の世界に存在する情報と、人間の脳における上記の表現の各々に対する情報表現をまず考える必要がある。今まで述べてきたように、後者には木構造が対応している。一方、前者には線形リストによる表現を思いつく。これは、計算機のメモリー上は、線形空間であることを思えば、ごく自然な対応付けであろう。そこで我々は情報の本質は線形リストであるが、これを木構造として扱う、という観点からプログラム変換のドメインに議論を持ち込むことにする。プログラム変換の入力となる宣言的プログラムは線形リストを利用した明快なものとし、変換結果は直接、木構造を扱う効率の良いプログラムをえられることを目的とする。なお以下では対象を実用上興味のある探索木の扱いに絞って議論を展開する。

2．宣言的なプログラム

宣言的なプログラムでは、1．に述べたヒントにより、木を一度、リストに展開し、そこでしかるべき操作を施し、そのリストを再び木に変換する。そこでまず木をリストに展開する述語について考えてみる。最も簡単なものは中島氏の著書Ｐｒｏｌｏｇに書かれている travers という述語である。

```
travers([],[])←.
travers(t(*l,*x,*r),*y) ←travers(*l,*ly),travers(*r,*ry),append(*ly,[*x|*ry],*y).      (1)
```

探索木 t(*l,*x,*r)を展開したリストのしかるべき位置へ要素 *a を挿入する述語 tins は travers を拡張して次のようになる。

```
tins([],[],[])←.                                                          (2.1)
tins(*a,[],[*a])←.                                                        (2.2)
tins([],t(*l,*x,*r),*y) ←tins([],*l,*ly),tins([],*r,*ry),append(*ly,[*x|*ry],*y).   (2.3)
tins(*a,t(*l,*x,*r),*y) ←*a<*x,tins(*a,*l,*ly),tins([],*r,*ry),
        append(*ly,[*x|*ry],*y).                                          (2.4)
tins(*a,t(*l,*x,*r),*y) ←*a≧*x,tins([],*l,*ly),tins(*a,*r,*ry),
        append(*ly,[*x|*ry],*y) .                                         (2.5)
```

(2.1) と(2.3) は traversと同じである。(2.4) と(2.5) は２分探索のアルゴリズムを表わしている。次にリストから木へ変換する述語 bltree を考える。これは append を逆に（generate type ）に用いれば次のように定義できる。

```
bltree([],[]) ←.                                                          (3.1)
bltree(*y,t(*l,*x,*r))←append(*ly,[*x|*ry],*y),bltree(*ly,*l),bltree(*ry,*r).   (3.2)
```

append がリスト *y から根に対応する *x と部分木に対応するリスト *ly,*ry を生成し、再帰的に木を組み立てていく。もちろん bltree は元の木との直接の対応はつかないが、木の性質を満たす木を次々と生成する。したがって我々が手続き的言語で記述した場合のような結果は何度か backtrack した後に得られる。 tins と bltree を組み合わせれば、木 *t に要素 *a を挿入した新たな木 *ta を得るプログラム ins は以下のようになる。

```
ins(*a,*t,*ta)←tins(*a,*t,*y),bltree(*y,*ta).                            (4)
```

3．変換

この節では、木へ要素を挿入するプログラム ins の変換過程を示すことによって、1．でのべたＨ１，Ｈ２などの Heuristic がどのような形に具現されるかを示そう。なお以下で頻繁に用いる unfold , fold の変換は、 Burstall, Darlington の提案したもの[10]を佐藤、玉木が Prolog に適用できる形に直したもの[5] である。まづ ins の第一 goal の tins を unfold すると、次の結果が得られる。

```
ins([],[],*a) ← bltree([],*a).                                            (5.1)
ins(*a,[],*b) ← bltree([*a],*b).                                          (5.2)
ins([],t(*a,*b,*c),*e)←
    tins([],*a,*f),tins([],*c,*g),append(*f,[*b|*g],*h),bltree(*h,*e).    (5.3)
ins(*a,t(*b,*c,*d),*f)←
    *a<*c,tins(*a,*b,*g),tins([],*d,*h),append(*g,[*c|*h],*i),bltree(*i,*f).   (5.4)
```

```
ins(*a,t(*b,*c,*d),*f)←
    *a ≧*c,tins([],*b,*g),tins(*a,*d,*h),append(*g,[*c|*h],*i),bltree(*i,*f).   (5.5)
```

(5.1),(5.2) からは bltree の unfold により次の終了条件がえられる。

```
ins([],[],[]) ← .                                                    (6.1)
ins(*a,[],t([],*a,[]))←.                                             (6.2)
```

(5.3) に関しては、H1の NIL 挿入による木の不変性の heuristic から次の結果を得る。

```
ins([],t(*a,*b,*c),t(*a,*b,*c)) ←.                                   (6.3)
```

この例によりH1が再計算の回避に役だっていることがわかる。次に (5.4)(5.5) の部分をリストを介さずに直接、木を操作するプログラムに変換する。(5.4) の bltree を unfold すると、

```
ins(*a,t(*b,*c,*d),t(*f,*g,*h)) ← *a<*c,tins(*a,*b,*i),tins([],*d,*j),
    append(*i,[*c|*j],*k),append(*l,[*g|*m],*k),
    bltree(*l,*f),bltree(*m,*h).                                     (6.4)
```

初めの append は *i と [*c|*j] を結合して *k を作り、二番目の append は逆に *k を分解して *l と [*g|*m] を生成している。H2によれば、再計算を避ける意味から一番目の append の引き数と二番目の append の引き数は同一のものとみなせる。すなわち *l を *i 、 *g を *c 、 *m を *j と名前変えできる。名前変えは clause 全体いに及んで行なう。この名前変えを済ませると、 append はもはや何等の意味ある仕事をしていない。そこで append を消去できる。この結果 (6.4) は次のようになる。

```
ins(*a,t(*b,*c,*d),t(*f,*c,*h)) ← *a<*c,tins(*a,*b,*i),tins([],*d,*j),
                                   bltree(*i,*f),bltree(*j,*h).       (6.5)
```

tins と bltree の各引き数の入出力の mode を考えれば、一番目の bltree は一番目の tins の直後に移動できる。すると tins , bltree という並びができる。これらは、 ins の元の定義 (4) に一致しており、 ins で fold 可能である。hold した結果は次のようになる。

```
ins(*a,t(*b,*c,*d),t(*f,*c,*d)) ← *a<*c,ins(*a,*b,*f),ins([],*d,*h).   (6.6)
```

ここで再びH1を用いると、二番目の ins は *d と *h が等しいことがわかる。したがって *h を *d と名前変えすれば、この ins も消去でき、次のような最終結果を得る。

```
ins(*a,t(*b,*c,*d),t(*f,*c,*d)) ← *a<*c,ins(*a,*b,*f).               (6.7)
```

(5.5) についても同様の変換を行なえば、次の結果が得られる。

```
ins(*a,t(*b,*c,*d),t(*b,*c,*f)) ← *a ≧*c,ins(*a,*d,*f).              (6.8)
```

(6.7),(6.8) は2分挿入のアルゴリズムを表わしている。最終的な手続き的プログラムは(6.1),(6.2),(6.3),(6.7),(6.8) の各 clause である。

4．例 （木からの要素の削除）

上記の方法で変換できる例として、木からの要素（node）の削除のプログラムを考えてみよう。宣言的なプログラムを作るために、まづ木 t(*l,*x,*r) から目的の要素 *a を削除したリスト *y を求める述語 tdelを定義する。

```
tdel([],[],[])←.                                                     (7.1)
tdel([],t(*l,*x,*r),*y) ← tdel([],*l,*ly),tdel([],*r,*ry),append(*ly,[*x|*ry]).   (7.2)
tdel(*a,t(*l,*x,*r),*y) ← *a<*x,
        tdel(*a,*l,*ly),tdel([],*r,*ry),append(*ly,[*x|*ry],*y).     (7.3)
```

```
tdel(*a,t(*l,*x,*r),*y) ← *a ≧*x,
        tdel([],*l,*ly),tdel(*a,*r,*ry),append(*ly,[*x|*ry],*y).        (7.4)
tdel(*a,t(*l,*a,*r),*y) ← tdel([],*l,*ly),tdel([],*r,*ry),append(*ly,*ry,*y).        (7.5)
```

(7.3),(7.4) は2分探索アルゴリズムを示し、(7.5) は *a に等しい要素を見つけた場合に、これを削除する部分である。したがって、木 *t から *a に等しい要素を削除した木 *ta を求める述語 del は次のようになる。

```
del(*a,*t,*ta)← tdel(*a,*t,*y),bltree(*y,*ta).        (8)
```

del の変換は ins の変換と似ている。ただし途中で新述語 dl を導入する。dl は変換の途中でててくる次の clause の body に対応して定義する。

```
tdel(*a,t(*b,*a,*c),t(*b,*i,*d)) ← tdel([],*c,[*i|*j]),bltree(*j,*d).
```

変換過程を全て示すと長くなるので、ここでは省略し、結果を示し、併せてその意味を説明する。

```
del([],[],[]) ←.        (9.1)
del([],t(*a,*b,*c),t(*a,*b,*c)) ←.        (9.2)
del(*a,t(*b,*c,*d),t(*e,*c,*d)) ← *a<*c,del(*a,*b,*e).        (9.3)
del(*a,t(*b,*c,*d),t(*b,*c,*e)) ← *a ≧*c,del(*a,*d,*e).        (9.4)
del(*a,t([],*a,*c),*c)←.        (9.5)
del(*a,t(*b,*a,[]),*b)←.        (9.6)
del(*a,t(*b,*a,*c),t(*b,*d,*e)) ← dl(*c,*e,*d).        (9.7)
dl(t([],*x,*r),*r,*x) ←.        (9.8)
dl(t(*l,*x,*r),t(*n,*x,*r),*z)← dl(*l,*n,*z).        (9.9)
```

(9.1) は終了条件である。(9.2) は heuristic H2による NIL 削除による木の不変性を表わしている。(9.3), (9.4) は2分探索アルゴリズムを表わしている。(9.5), (9.6) は探索によって見つけた *a に子供がひとつしかない場合の処理で、単に *a を木から削除することをあらわしている。(9.7) は探索によって見つけた *a に子供がふたつ付いている場合に、新述語 dl を呼ぶものである。dl(*t,*r,*x) は木 *t の左部分木の最右要素 *x を取り出し、残りの部分木 *r を返すものである。(9.8) は最右の要素が発見された場合であり、(9.9) は左部分木の右側をたどる操作を表わしている。このプログラムでは変換のステップ数は新述語の導入により若干増えているが、 ins の場合とほとんど同じコースで結果を得ることができる。

5．条件付きの木への拡張

以上述べてきた方法をバランス木やＡＶＬバランス木、さらにはＢ木の扱いに拡張する方法について簡単に触れておこう。これらの木は木の性質自体に条件がついているのだが、このような条件は、この方法の場合、木を生成する述語 bltree に条件を満たすかどうかを判定する goal の形で付加すればよい。

```
bltree([],[],*n)←.
bltree(*z,t(*l,*x,*r,*n),*n)← append(*l1,[*x|*r1],*z),
        bltree(*l1,*l,*m),bltree(*r1,*r,*k),test(*m,*k,*n).        (10)
```

木の４番目の *n は木の高さや構成要素の数などの、木の性質を表わしている。ここで問題になるのは test に fail して backtrack したとき、つぎにいかなる木構造を生成すればよいかである。これは人間がアリゴリズムを考えるときに種になる最も本質的な部分である。次の解となる木を生成する方法として我々が

よく知っているのは、つぎのような heuristic である。

(1) 一要素の移動： 左（右）部分木から最右（左）の要素を右（左）部分木の最左（右）へ移す。この heuristic は完全バランス木を扱うアルゴリズムを考える際に用いる。

(2) append の結合法則： bltree が生成する二番目の解を append の結合法則を適用することにより求める。この heuristic はAVL-バランス木に対するアルゴリズムの本質的部分である。

(3) 2分割： 多分木を考える際には、多数の要素を含む node を2分割する heuristic が有用である。B－木に関するアルゴリズムのベースになる。

これらはいずれも広い意味での append の結合法則とみなせる。もちろん、このほかにも種々の heuristic があるだろうが、それを発見することは、人間の創造の能力にほとんど近い（あるいは等しい）と考えられ、自動的に行なうことは現在のところ困難である。

さて上記のような heuristic を利用して2番目以降の解の候補が求まると、 bltree を場合分け(case split)して、抽象的に書くと次のような形のプログラムがえられる。

```
bltree(*z,f(*x))← append(*),append(**),bltree,bltree,test1(*x).
bltree(*z,g(*x))← append(*1),append(**1),bltree,bltree,test2(*x).
      :                :                                          (11.1)
```

bltree をこのように変換した後、 tins あるいは tdel の append との組み合わせに、H2を適用し、 append を消去する。この結果、(11.1) に対応して次のようなプログラムを得る。－H3.

```
p(*a,*b,f(*z))← q,r(*a,*b,*z),test1(*z).
p(*a,*b,g(*z))← q,r(*a,*b,*z),test2(*z).
   :                :                                             (11.2)
```

通常、r(*a,*b,f(*z)) は要素 *a を木 *b に挿入／削除して新たな木 f(*z) を得る述語であり、計算量の大きなものである。そこで r(*a,*b,f(*z)) の再計算を避けるために次の変換を導入する。－H4.

(11.2) において、各assertion 毎に異なる test1, test2,..の部分をまとめた新述語 s を導入し、(ii.2) を以下のように変換する。

```
p(*a,*b,*x) ← q,r(*a,*b,*z),s(*z,*x).
s(*z,f(*z)) ← test1(*z).
s(*z,g(*z)) ← test2(*z).
 :           :                                                    (11.3)
```

これにより、r(*a,*b,*x) の計算は一回しか行なわれなくんる。明かにこの変換は、(11.2) の場合分けが可能な場合をすべて尽くしていれば、等価な変換である。以上の変換コースにより、木構造を扱うアルゴリズムの効率化が達成される。

6．例 （ＡＶＬ－バランス木への要素の挿入）

５．で述べた変換の一例としてＡＶＬ－バランス木への要素の挿入のアルゴリズムを変換により導いてみよう。ＡＶＬ－バランス木は左右の部分木の高さの差が2より小さいという性質を持つ2分木である。この性質を表現するためには、 bltree をつぎのように変える。

```
bltree([],[],0).
bltree(*z,t(*l,*x,*r,*n),*n)← append(*i1,[*x|*r1],*z),
        bltree(*l1,*l,*m),bltree(*r1,*r,*k),test(*m,*k,*n).      (12.1)
```

ここで test は部分木の性質に関する条件をテストするための述語で、左右の部分木の高さの差が２より小さい場合にのみ成功する。

```
test(*m,*k,*n)← *m>*k,*d is *m-*k,2>*d,*n is *d+*m.
test(*m,*k,*n)← *m<*k,*d is *k-*m,2>*d,*n is *d+*k
test(*m,*m *n)← *n is *m+1 .                                    (12.2)
```

これらと tins を組み合わせれば、AVL-バランス木 *t へ要素 *a を挿入し、高さ *n のAVL-バランス木 *ta を得る述語 brins は次の形になる。

```
brins(*a,*t,*ta,*n) ← tins(*a,*t,*y),bltree(*y,*ta,*n).        (12.3)
```

以下に brins から効率の良いプログラムを導出する変換について述べる。まづ、(12.3) を unfold することによって、次の終了条件を得る。

```
brins([],[],[],0).                                              (13.1)
brins(*a,[],t([],*a,[],*n),*n)← test(0,0,*n).                  (13.2)
```

また heuristic Ｈ１ により NIL 挿入の場合の扱いを得る。

```
brins([],t(*l,*x,*r,*n),t(*l,*x,*r,*n),*n).                    (13.3)
```

(12.3) を unfold した残りの部分は次のものである。

```
brins(*a,t(*b,*c,*d,*e),*f,*g)← *a<*c,tins(*a,*b,*h),tins([],*d,*i),
        append(*h,[*c|*i],*j),bltree(*j,*f,*g).                 (13.4)
brins(*a,t(*b,*c,*d,*e),*f,*g)← *a ≧*c,tins([],*b,*h),tins(*a,*d,*i),
        append(*h,[*c|*i],*j),bltree(*j,*f,*g).                 (13.5)
```

ここでは(13.4) の変換について述べる。(13.5) については対称的に変換できるので省略する。(13.4) の最初の tins と bltree を unfold すると次のようになる。

```
brins(*a,t(t(*b,*c,*d,*e),*f,*g,*n),t(*i,*j,*k,*l),*i)← *a<*f,*a<*c,
        tins(*a,*b,*m),tins([],*d,*p),append(*m,[*c|*n],*o),tins([],*g,*p),
        append(*o,[*f|*p],*q),append(*r,[*j|*s],*q),
        bltree(*r,*i,*t),bltree(*s,*k,*u),test(*t,*u,*l).       (13.6)
```

ここでＨ２により*r←*o,*j ←*f,*s ←*p と名前変えをする。tins 側のappend でさらに展開されている *o を調べるために一番目の bltree をもう一回 unfold し同様にＨ２によって append にでてくる変数を名前変えすると、次のようになる。

```
brins(*a,t(t(*b,*c,*d,*e),*f,*g,*h),t(t(*i,*c,*k,*l),*f,*m,*n),*n)←
        *a<*f,*a<*c,tins(*a,*b,*o),tins([],*d,*p),append(*o,[*c|*p],*q),tins(*q,[*f|*r],*s),
        append(*o,[*c|*p],*q),bltree(*o,*i,*v),bltree(*p,*k,*w),test(*v,*w,*l),
        bltree(*r,*m,*y),test(*l,*y,*n)                         (13.7)
```

ここで２番目の解の候補を求めるためにＨ３により場合分けを行なう。この場合は append の結合法則による。すなわち、(13.7) の３、４番目の append を次のように変形したものを場合分けする。

```
append(*o,[*c|*q],*s),append(*p,[*f|*r],*q)->append(*q,[*f|*r],*s),append(*o,[*c|*p],*q)
```

この変換に伴って (13.7) の head に出力として表われる木も次のように変換する。

```
t(*i,*c,t(*k,*f,*m,*l),*n)  この後 append を消去すると次のふたつの clause がえられる。
brins(*a,t(t(*b,*c,*d,*e),*f,*g,*h),t(t(*i,*c,*k,*l),*f,*m,*n),*n)← *a<*f,*a<*c,
        tins(*a,*b,*o),tins([],*d,*p),tins([],*g,*r),bltree(*o,*i,*v) ,bltree(*p,*k,*w),
```

```
        bltree(*r,*m,*y),test(*v,*w,*l),test(*l,*y,*n).                    (13.8)
brins(*a,t(t(*b,*c,*d,*e),*f,*g,*h),t(*i,*c,t(*k,*f,*m,*l),*n),*n)← *a<*f,*a<*c,
        tins(*a,*b,*o),tins([],*d,*p),tins([],*g,*r),bltree(*o,*i,*v), bltree(*p,*k,*w),
        bltree(*r,*m,*y),test(*w,*y,*l),test(*l,*v,*n).                    (13.9)
```

変数の入出力のモードに関する知識によって、tins と bltree を適当に入れ替えれば、brins の、元の定義 (12.3) によって fold できる tins,bltree のペアが表われる。これらをすべて fold した後、NIL 挿入による木の不変性によって、次のような変換をおこなう。brins([],*a,*b,*c)← brins([],*a,*a,*c) これらの結果は次のようになる。

```
brins(*a,t(t(*b,*c,*d,*e),*f,*g,*h),t(t(*i,*c,*d,*l),*f,*g,*n),*n)← *a<*f,*a<*c,
        brins(*a,*b,*i,*v),brins([],*d,*d,*w),brins([],*g,*g,*y) ,
        test(*v,*w,*l),test(*l,*y,*n).                                     (13.10)
brins(*a,t(t(*b,*c,*d,*e),*f,*g,*h),t(*i,*c,t(*d,*f,*g,*l),*n),*n)←  *a<*f,*a<*c,
        brins(*a,*b,*i,*v),brins([],*d,*d,*w),brins([],*g,*g,*y),
        test(*w,*l,*l),test(*v,*l,*n).                                     (13.11)
```

これらが単一 LL- 回転のアルゴリズムである。brins([],*a,*a,*c) のパターンのgoal は(13.2) と一回だけ unify されるだけなので、計算量は少ない。最後にＨ４によって brins の再計算を避けるための新述語 sll を導入すれば変換が完成する。結果は次のとうりである。

```
brins(*a,t(t(*b,*c,*d,*e),*f,*g,*h),*x,*n)← *a<*f,*a<*c, brins(*a,*b,*i,*v),
        brins([],*d,*d,*w), brins([],*g,*g,*y),sll(*i,*c,*f,*g,*v,*m,*y,*n,*x).   (13.12)
sll(*i,*c,*d,*f,*g,*v,*m,*y,*n,t(t(*i,*c,*d,*k),*f,*g,*n))←test(*v,*m,*k),test(*k,*y,*n).
sll(*i,*c,*d,*f,*g,*v,*m,*y,*n,t(*i,*c,t(*d,*f,*g,*k),*n))←test(*m,*y,*k),test(*k,*v,*n).
                                                          (13.13) , (13.14)
```

brins をunfold した結果、 *a ≧*f,*a<*c になる場合は、tins と brins を上記の場合よりもう一回多く unfold し、さらに append の結合法則を２回適用すると、それ以外の部分は上記の場合とほとんど同様に変換して、二重LR- 回転のアルゴリズムが得られる。また、RL- , RR- 回転のアルゴリズムは (13.5) を上記と同様に変換することによって得られる。また、(13.1),(13.2) 以外の終了条件は unfold のみでてくる。最終的な結果を全て載せると長くなるので省略する。

７．おわりに

木構造を扱うアルゴリズムをプログラム変換によって導出する手法についてのべた。特に木を一度リストに展開する方法により、木に関するポインタの付け替えの操作をリスト上における append の結合法則に置き換えて考えられる。多分、結合法則の方が直感的にわかりやすいようにおもわれる。将来の課題は、人間がアルゴリズムを発案する際に使っている heuristic の構造の解明であり、これができれば、真の意味で自動プログラミングに近づくと思っている。当面は多分木についてこの方法を適用する方向を検討していく。その結果については近いうちに発表したい。

［謝辞］
本研究にあたっては電総研の佐藤氏の”変換を進めるにはデータ構造が重要だ。”という一言がヒントになった。厚く感謝する。また、情報大の平賀氏の”人間の思考の基本のひとつは多次元の思考を一次元に変換

する点だ。”という考えも非常に参考になった。電総研の中島氏を中心とする人工知能の勉強会”AIUEO”の discussion はいつもながら良いsuggestion である。深く感謝する。また本研究に必須の tool である会話型のプログラム変換システム TRANS を開発してくれた横浜国大工学部情報工学科中川研究室の中村直人君には大変お世話になった。TRANS がなければ、長大なプログラムの変換と変換結果のプログラムの動作確認はこれほど短期間にはできなかっただろう。最後に本研究は文部省科学研究費（特定研究－多次元知識情報処理）に予算のバックグランドを得たことを記し関係各位に感謝の意を顕はしたい。

［参考文献］

[1] Partsch,etal.,"Program Transformation Systems",ACMComp.Surveys,Vol.15,no.3,1983

[2] Hogger,C.J.,"Derivation of Logic Programs",JACM Vol.28 No.2, 1981

[3] Bible,W.,"Syntax-Directed,Semantic-Supported Program Synthesis",Artficial Intelligence 14, North-Holland, 1980

[4] Sato,T.,"Transformational Logic Program Synthesis", Proc. of FGCS'84, 1984

[5] H.Tamaki,T.Sato,"Unfold/Fold Transformation of Logic Programs",2'nd logic programming conference,1984

[6] Tarnlund,S-A.,Hansson,A.,"Program Transformation by Data Structure Mapping",LOGIC PROGRAMMING, Academic-Press, 1982

[7] Kahn,K.M.,"A Partial Evaluation of Lisp Programs written in Prolog", 1'st International Logic Program Conference, 1982

[8] 竹島、他 ”PROLOGソースレベルオプティマイザ”,Proc. of The Logic Programming Coference'84, 19841

[9] 中村、他 ”Heuristic を用いたPrologプログラムの効率化変換”,Proc. of The Logic Programming Conferecnce'84, 1984

[10] Reddy,U.S.,"Transformation Of Logic Programs into Functional Programs", International Symposium on LOgic Programming, Atlantic City ,1984

[11] Burstall R.M,Darlington J.,"A Transformation System for Developing Recursive Programs",JACM Vol.24,No.1, 1977

[12] Darlington,J.,"The Synthesis of Implementations for Abstract Data Types", Computer Program Synthesis Methodologies Proc. of NATO Advanced Study Institute, 1981

[13] Darlington,J.,"An Experimental Program Transformation and Synthesis System", Artificial Intelligence Vol.16 ,1981

[14] Wand,M.,"Continuation-Based Program TRansformation Strategies",JACM Vol.27 No.1, 1980